Konstruktion og implementering af interaktivt theorem proving system

Willy Mathiesen

DAIMI PB - 4
Maj 1972

DATALOGISK AFDELING
AARHUS UNIVERSITET
Ny Munkegade, Bygn. 540
8000 Aarhus C, Denmark

Matematisk Institut
Aarhus Universitet

29. april 1972

NATURVIDENSKABELIG EMBEDSEKSAMEN

med hovedfag i matematik

2. del

Skriftlig opgave til besvarelse i løbet af 4 uger

Stud. scient. Willy Mathiesen

"Konstruktion og implementering af interaktivt theorem proving system"

Opgaven stillet: 29. april 1972

Opgaven afleveres: 26. maj 1972

Brian H. Mayoh

Ebbe Thue Poulsen

" KONSTRUKTION OG IMPLEMENTERING AF INTERAKTIVT THEOREM PROVING SYSTEM "

Willy Mathiesen.

maj 1972.

INDHOLD.

1. THEOREM PROVING OVERSIGT side 1.

2. BESKRIVELSE AF ET KONKRET THEOREM PROVING SYSTEM side 11.

3. TRÆMETODEN side 16.

4. DOKUMENTATION AF PROGRAM side 27.

5. EKSEMPLER side 49.

6. VURDERING OG KOMMENTAR side 58.

APPENDIX A : PROGRAMUDSKRIFT.

APPENDIX B : BRUGERVEJLEDNING.

AFSNIT 1

THEOREM PROVING OVERSIGT

Det problem, der behandles i dette arbejde, er automatisk theorem proving i prædikatkalkylen, og jeg vil i dette indledende afsnit give en kort oversigt over problemstillingen og de metoder og resultater, man indtil nu har opnået inden for denne del af artificial intelligence forskningen.

Problemstilling

Næsten alle hidtidige arbejder har indskrænket sig til at betragte 1. ordens prædikatkalkyle med funktioner og evt. med lighedstegn.*) I et sådant system forstår man ved en sætning en velformet formel uden frie variable, d.v.s. en sætning får ved en given interpretation sandhedsværdien sand eller falsk. En sætning siges at være gyldig, hvis den får værdien sand ved alle mulige (måske uendelig mange) interpretationer. Hvis $A=\{A_1, A_2, \ldots, A_n\}$ er en mængde af sætninger, og $A_1, A_2, \ldots, A_n$ alle er sande i en given intrepretation, siges denne interpretation at tilfredsstille mængden A eller at være en model for A. Hvis der ikke findes nogen interpretation, der tilfredsstiller A, siges mængden at være unsatisfiable. Dersom alle interpretationer, der tilfredsstiller mængden A, også tilfredsstiller sætningen B, siges B at være en logisk konsekvens af A (notation: $A \vdash B$). Det centrale problem er nu: Hvordan afgør man, om en given sætning er en logisk konsekvens af en mængde givne forudsætninger? Church's Theorem (1936) giver det vigtige resultat, at prædikatkalkylen er "undecidable", d.v.s. der findes ingen mekanisk procedure, der for en vilkårlig mængde forudsætninger A og en vilkårlig sætning B kan afgøre om $A \vdash B$ og levere et resultat i form af ja eller nej. Derimod kan man udmærket konstruere algoritmer, der, såfremt B følger af A, vil give svaret ja i løbet af endelig tid, men som ikke altid vil give et svar, hvis B ikke følger af A, idet algoritmen i så fald måske fortsætter uendelig længe. Når en sådan algoritme har kørt en vis tid uden at give noget svar, er man altså ikke istand til at afgøre, om det er fordi

*) : en præcis definition af prædikatkalkylen gives i afsnit 3.

teoremet ikke gælder, eller om algoritmen ville have givet svaret ja, hvis den havde fået lov at fortsætte.

Omkring 1960 begyndte man at beskæftige sig med problemerne omkring implementering af sådanne algoritmer på en datamat. Allerede 1957 havde Newell, Shaw og Simon publiceret et program, Logic Theorist, der kunne bevise teoremer i sætningskalkylen, og nu fulgte arbejder af bl.a. Wang, Gilmore, Davis og Putnam, der forsøgte den væsentlig vanskeligere opgave at implementere bevisprocedurer for teoremer i prædikatkalkylen. Disse første theorem-provere havde selvsagt en meget begrænset formåen, og det var først med udviklingen af Robinsons resolutionsprincip (1965), at man fik et hjælpemiddel, der lovede mulighed for konstruktion af theorem-provere, der kunne bevise teoremer af en rimelig kompliceret struktur. Siden da har emnet været genstand for en enorm interesse og en lang række arbejder, næsten alle baseret på resolutionsprincippet, har beskæftiget sig dels med teoretiske resultater og dels med konkrete implementationer.

Resolution/paramodulation

Resolutionsprincippet er en inferensregel, der ud fra to givne sætninger evt. giver en tredje, som logisk følger af de to givne. For at kunne anvende resolutionsprincippet må alle sætningerne først omskrives på klausulform (kvantorfri, konjunktiv normalform) d.v.s. som en konjunktion af klausuler, der hver består af en disjunktion af literals, d.v.s. atomare formler eller negationen af atomare formler. Ved denne omskrivning dimineres existenskvantorer ved indførelse af Skolem-funktioner, og alkvantorerne forsvinder, idet det underforstås, at alle variable er bundet af en foranstillet alkvantor.

En substitution $\Theta = \{v_1 \to t_1, v_2 \to t_2, \ldots, v_n \to t_n\}$ i en klausul L består i, at alle forekomster af variablen v_i erstattes af termen t_i ($i=1,2, \ldots, n$), hvorved fås klausulen $\Theta(L)$, der logisk følger af L, da alle variable i L jo er bundet af en underforstået alkvantor. I det følgende vil $\Theta(L)$ blive kaldt en variant af L.

Det generelle resolutionsprincip kan nu formuleres på følgende måde:

Hvis $L = l_1 \vee l_2 \vee \ldots \vee l_n$ og
$M = m_1 \vee m_2 \vee \ldots \vee m_k$ er givne
klausuler, og Θ og Ψ er substitutioner,
således at $\Theta(l_1) = \neg\Psi(m_1)$, defineres en
ny klausul
$R(L, M) = \Theta(l_2) \vee \ldots \vee \Theta(l_n) \vee \Psi(m_2) \vee \ldots \vee \Psi(m_k)$.

R(L ,M) kaldes en resolvent af L og M og følger logisk af disse.

Det understreges, at L og M kan have mange forskellige resolventer, idet Θ og Ψ kan vælges på forskellige måder. Hvis man har givet en mængde af klausuler, og man ud fra to af disse v.hj.a. resolutionsprincippet kan danne den tomme klausul (NIL), må den givne mængde af klausuler være unsatisfiable. R(L,M) = NIL kan n lig kun forekomme, hvis en variant af L og en variant af M er lig med henholdsvis et literal og dets negation.

Resolutionsprincippet kan anvendes til at bevise, at sætning B følger logisk af en mængde af sætninger A, hvilket er ækvivalent med at vise, at mængden $A \cup \{\neg B\}$ er unsatisfiable:

Først omskrives $A \cup \{\neg B\}$ til en mængde S af klausuler.
Dernæst dannes mængden $\Re(S) = S \cup \{$alle mulige resolventer af par af klausuler i $S\}$.
Dernæst dannes mængderne $\Re^2(S) = \Re(\Re(S))$, $\Re^3(S)$ osv.
Hvis en af disse mængder $\Re^i(S)$ indeholder NIL, gælder det altså, at $\Re^i(S)$ og dermed S er unsatisfiable, dvs. $A \vdash B$.

Resolutionsprincippets anvendelighed hviler nu tungt på flg. fundamentale sætning (Robinson, 1965) [2]:

Hvis en endelig mængde S af klausuler er unsatisfiable, vil der findes et endeligt tal i, så at $NIL \in \Re^i(S)$.

Metoden er altså fuldstændig i den forstand, at såfremt $A \vdash B$, vil dette altid blive afsløret af den skitserede algoritme i løbet af endelig tid.

Såfremt det logiske system omfatter 1. ordens logik med lighedstegn, kræves der en speciel behandling af lighedsprædikatet. Robinson og Wos har vist, at ovennævnte algoritme også fungerer i dette tilfælde, hvis man foruden de resolventer, der kan deduceres fra S, også danner de paramodulanter, der fremkommer på flg. måde:

> Givet klausulerne
> $L = (t_1 = t_2) \vee l_2 \vee \ldots \vee l_n$ og
> $M = m_1 \vee m_2 \vee \ldots \vee m_k$
> hvor t_1 og t_2 er termer, og m_1 indeholder en forekomst af termen t. Hvis der findes substitutioner Θ og Ψ, så at $\Theta(t_1) = \Psi(t)$, så defineres en ny klausul
> $\Theta(l_2) \vee \ldots \vee \Theta(l_n) \vee m_1{}^* \vee \Psi(m_2) \vee \ldots \vee \Psi(m_k)$,
> hvor $m_1{}^*$ er fremkommet ved i $\Psi(m_1)$ at erstatte en forekomst af $\Psi(t)$ med $\Theta(t_2)$. Denne nye klausul kaldes en paramodulant fra L ind i M og følger logisk af L og M.

Søgestrategier

Den ovenfor omtalte algoritme svarer til en bredde - først søgning efter NIL, idet man successivt danner alle klausuler i dybde 0 (S), dybde 1 ($\mathfrak{R}(S)$), dybde 2 ($\mathfrak{R}^2(S)$) osv. P.gr.a. den explosionsagtige vækst i antallet af klausuler, vil en lige-ud-af-landevejen implementering af denne algoritme være praktisk håbløs, og interessen har derfor koncentreret sig om at udvikle søgestrategier, der reducerer antallet af overflødige resolutioner/paramodulationer. Mens man endnu ikke har fundet virkelig effektive metoder til håndtering af paramodulation, og meget få theorem-provere derfor omfatter lighedstegnet, er der blevet udviklet en lang række strategier, der effektivt nedskærer antallet af resolutioner.

En stor klasse af strategier kan karakteriseres ved, at de kun tillader resolutioner mellem par af klausuler L og M, såfremt disse tilfredsstiller en til strategien hørende betingelse P(L,M). Hvis $\mathfrak{R}^i{}_p(S)$ betegner den delmængde af $\mathfrak{R}^i(S)$, der vil blive dannet ved en sådan strategi, kan $\mathfrak{R}^i{}_p(S)$ defineres rekursivt:

$\mathfrak{R}^{0}{}_{p}(S) = S.$

$\mathfrak{R}^{n+1}{}_{p}(S) = \{K | K$ er en resolvent af L og M $\wedge$

$L, M \in \mathfrak{R}^{n}{}_{p}(S) \wedge P(L,M)\} \cup \mathfrak{R}^{n}{}_{p}(S).$

Et eksempel på en sådan strategi er den hyppigt anvendte Set of Support-strategi, der før starten kræver, at en delmængde K af mængden S af startklausuler udvælges på en sådan måde, at $S \setminus K$ formodes at være satisfiable. Mængden K siges at have "support". Den til strategien hørende betingelse P er da:

$P(L,M) \Leftrightarrow$ L har support $\vee$ M har support.

Hver ny klausul, der dannes, siges ligeledes at have support. Denne strategi har en oplagt anvendelse, når man skal bevise teoremet $A \vdash B$, idet de klausuler, der hidrører fra $\neg B$, kan vælges som havende support. Dette svarer intuitivt til at forbyde deduktioner udelukkende på grundlag af præmisserne, når man søger efter en modstrid.

Denne strategi er fuldstændig i den forstand, at hvis mængden S er unsatisfiable, og K er valgt, så at $S \setminus K$ er satisfiable, så vil der findes et endeligt tal i, så at $NIL \in \mathfrak{R}^{i}{}_{p}(S)$.

Det gælder for denne strategi og andre af samme type, at de giver anledning til væsentlig snævrere men til gengæld dybere søgninger, idet den simplest mulige deduktion af NIL almindeligvis vil blive udelukket af den restriktion, som strategien indebærer. Værdien af en sådan strategi skal altså måles ved den evt. reduktion af det totale arbejde, heri medregnet både antal resolutioner og arbejdet med at teste om to klausuler opfylder betingelsen P. Det er således absolut ikke sikkert, at den mest restriktive strategi også er den bedste. Luckham [3] gør opmærksom på, at mens det i begyndelsen var pladsproblemet, der motiverede, at man begyndte at udvikle pladsbesparende strategier, er man nu kommet i den situation, at man har fået en række strategier, der er så forfinede, at de bruger næsten al tiden på at spare plads, så tidsproblemet nu er blevet et ligeså alvorligt problem.

En anden klasse af strategier er de såkaldte ordningsstrategier, der er karakteriseret ved, at de ikke udelukker nogle resolutioner, men hver gang en resolution skal udføres, vælges de indgående klausuler ifølge en evalueringsfunktion, hvis værdier afhænger af klausulernes komplexitet i en eller anden forstand.

Et eksempel på en strategi af denne type er Unit Preference-strategien, der giver højeste prioritet til resolutioner, hvori indgår klausuler med kun ét literal, og i øvrigt prioriterer de resolutioner højest, der resulterer i nye klausuler med færrest mulige literals. Dette er begrundet i, at målet jo er at frembringe den tomme klausul.

Hvis man ønsker at bevare fuldstændigheden, må en sådan strategi kombineres med en dybdegrænse N, der forhindrer, at der dannes klausuler fra mængden $\mathcal{R}^{N+1}(S)$, før alle mængderne $\mathcal{R}(S)$, $\mathcal{R}^2(S)$, ..., $\mathcal{R}^N(S)$ er "fyldt op". Derved undgår man, at algoritmen løber løbsk i en nytteløs kæde af resolutioner.

Fuldstændighed/effektivitet

En meget stor del af arbejderne inden for theorem proving har koncentreret sig om at bevise fuldstændigheden af de anvendte strategier, og der har været en tendens til kun at indbygge strategier med denne egenskab i de programmer, der hidtil er lavet. Derimod ved man meget mindre om de forskellige strategiers fortrin og mangler m.h.t. effektivitet og praktisk anvendelighed. De fleste strategier er til god hjælp ved beviset for visse teoremer, mens de stillet over for andre problemer ikke er til nogen hjælp og undertiden endda komplicerer arbejdet unødigt. Det er meget vanskeligt på forhånd at teoretisere sig til hvilke strategier, der er bedst egnede til hvilke problemtyper, og det er et område, hvor der er behov for konkrete experimenter, således at man kan få en række empiriske resultater, der forhåbentlig kan give indsigt i dette samspil. Alt tyder altså på, at man ikke kan forvente at finde én strategi, som egner sig lige godt til alle problemer, og dette peger i retning af, at man bør satse på interaktive systemer, der giver brugeren mulighed for selv at specificere den ønskede strategi. Et sådant system er beskrevet udførligere i afsnit 2.

Det er klart, at fuldstændighed af en strategi er en teoretisk meget ønskelig egenskab, men meget kunne tyde på, at den énsidige interesse for netop denne egenskab har været en medvirkende årsag til de relativt små forbedringer af theorem-proveres effektivitet i de seneste år, små i forhold til den interesse og arbejdsinsats, der har været ofret på emnet. L.M. Norton [4] diskuterer dette problem i indledningen til beskrivelsen af en konkret theorem-prover og udtrykker sit synspunkt således: "The relevance of a result showing the theoretical superiority of one strategy over another lies in whether or not the strategy can be used to improve the performance of a program on actual problems. --- The inability of existing programs to attain significant levels of ability without sacrificing completeness, either by altering basic algorithms or by adding special-purpose heuristics, tends to convince us that the worth of many theoretical results involving completeness either lies wholly in the future or is purely academic."

Af denne grund har han forsynet sit program med en række heuristiske "tricks", og specielt har han interesseret sig for at udvikle heuristikker, der kan finde anvendelse på lighedstegn. Et eksempel på en sådan heuristik er følgende:

Hvis $L = (t_1 = t_2) \vee l_2 \vee \ldots \vee l_n$ er en klausul, vil en paramodulation fra L ind i en klausul M bevirke, at en variant af t_1 i M vil blive erstattet af en variant af t_2 eller omvendt. Nu har man en intuitiv fornemmelse af, at nogle paramodulationer simplificerer de fremkomne klausuler, mens andre øger komplexiteten. Hvis $L = (x\,x^{-1} = e) \vee l_2 \vee \ldots \vee l_n$, er det f.eks. naturligt at opfatte en udskiftning af $x\,x^{-1}$ med e som en simplifikation og en udskiftning af e med $x\,x^{-1}$ som en komplikation. Brugeren af programmet har derfor mulighed for på forhånd (ikke interaktivt) at erklære for hvert lighedstegn i startklausulerne hvilke udskiftninger, der er simplifikationer, og hvilke der er komplikationer. Denne information udnyttes så, når programmet skal afgøre hvilken paramodulation, der skal udføres.

Selv om Nortons program trods adskillige sådanne heuristiske metoder ikke har en væsentlig større formåen end flere andre theorem-provere, er der dog næppe tvivl om, at interessen i de kommende år vil koncentrere sig om udvikling af sådanne ufuldstændige men forhåbentlig mere effektive metoder.

Når man betænker, at menneskelig problemløsning vel kun meget sjældent betjener sig af en fuldstændig metodik, kan det jo i hvert fald synes urimeligt at interessere sig så meget for at indbygge en fuldstændighed i et artificial intelligence projekt, som man hidtil har gjort.

Anvendelser

Det sidst nævnte problem hænger naturligvis nøje sammen med, hvilket formål man har med at konstruere en theorem-prover.

Grunden til, at theorem proving har fået en så fremtrædende plads inden for artificial intelligence, er vel, at netop matematisk bevisførelse er en opgave, som typisk anses for at kræve intelligens og intuition. Ved at beskæftige sig med en formaliseret løsning af sådanne opgaver håber man også at få indsigt i de ofte ubevidste metoder, som mennesker betjener sig af, når de løser lignende opgaver, og derigennem evt. få inspiration til heuristiske programmeringsmetoder til mere generel problemløsning.

Men derudover er man naturligvis også interesseret i at anvende programmerne til mere praktiske formål. Mest oplagt er det at forvente, at man her får et værktøj, v.hj.a. hvilket man måske kan bevise nye matematiske teoremer. De hidtidige resultater viser, at existerende theorem-provere har en kapacitet, der f.eks. tillader dem at bevise teoremer og standardøvelsesopgaver inden for elementær algebra og talteori, og det er i ganske enkelte tilfælde rapporteret, at man har fundet en løsning til et hidtil åbent problem. Det kan således udmærket tænkes, at man i fremtiden ad denne vej kan få nye ikke-trivielle resultater.

Et felt, hvor udsigterne til praktiske resultater er mindre lovende, er anvendelse af theorem proving i forbindelse med question answering systemer. Hvis man i et information-retrieval system har opbygget en database, er det ønskeligt, at man ikke alene har adgang til den information, der er explicit lagret, men også til den information, der kan deduceres ud fra databasen. Selv om prædikatkalkylen er en velegnet notationsform for mange praktiske emneområder, er datamængden i de fleste realistiske anvendelser så stor, at de nuværende bevisalgoritmer ikke er tilstrækkelig effektive.

Derimod har man haft stor succes med at benytte theorem proving i forbindelse med databaser for specielle simple problemområder, hvor antallet af axiomer og relationer er rimeligt lille, først og fremmest i forbindelse med robotprojekter . I forbindelse med Stanfords robotprojekt har man udviklet et planlægningsprogram STRIPS [5], der benytter theorem proving til at bestemme den sekvens af handlinger, som robotten skal udføre for at nå et ønsket mål. STRIPS indeholder en model af robottens ydre verden i form af en række sætninger i prædikatkalkylen, f. eks. AT (robot, 3, 5), der betyder, at robotten befinder sig på stedet med koordinaterne (3, 5), og

$$\forall\, w, x1, y1, x2, y2 : [AT(w,x1,y1) \wedge \neg((x1=x2) \wedge (y1=y2))] \rightarrow \neg AT(w,x2,y2)$$

der udtrykker, at et objekt ikke kan befinde sig på to forskellige steder. Et mål for robotten kan også formuleres som en sætning i prædikatkalkylen f. eks. NEXTTO (box 1, box 2), der betyder, at box 1 og box 2 skal flyttes sammen. Hver handling, som robotten kan udføre, er beskrevet v. hj. a. en liste over de sætninger, der skal være opfyldt i en model, for at handlingen kan bringes i anvendelse, samt en liste over de ændringer i modellen, som handlingen vil medføre. Theorem-proverens opgave er da at bevise, at det ønskede mål kan deduceres ud fra den givne model og ud fra et sådant bevis at extrahere den sekvens af handlinger, der fører til målet, og overlade denne plan til den del af robotten, der sørger for udførelsen af de forskellige handlinger. Automatisk theorem proving bliver altså en meget væsentlig del af et robotprojekt.

Endelig skal det nævnes, at theorem proving har fundet anvendelse inden for området automatisk programverifikation, og i relation til dette er der i de seneste år fremkommet arbejder, der diskuterer anvendelser vedr. automatisk programskrivning, men en behandling af hele dette problemkomplex falder uden for rammerne af denne oversigt.

AFSNIT 2

BESKRIVELSE AF ET KONKRET THEOREM PROVING SYSTEM

Det hidtil mest ambitiøse og mest udviklede theorem proving program er beskrevet af Allen og Luckham i [6]. Dette program er under stadig udvikling ved Stanford University, og den version, der her skal beskrives, er tænkt som kærnen i et fremtidigt endnu mere omfattende og fleksibelt system.

Programmets struktur

Programmets struktur kan anskueliggøres v.hj.a. rutediagrammet i figur 1.

Gennem en række interaktive kommandoer starter brugeren af systemet med at indlæse startklausulerne og initialisere en række programvariable, som bestemmer hvilken kombination af udvalgs- og editeringsstrategier, der ønskes anvendt. En løkke i beregningen består da af følgende trin:

1. Ved hjælp af de specificerede udvalgsstrategier udvælges to klausuler blandt de hidtil lagrede.

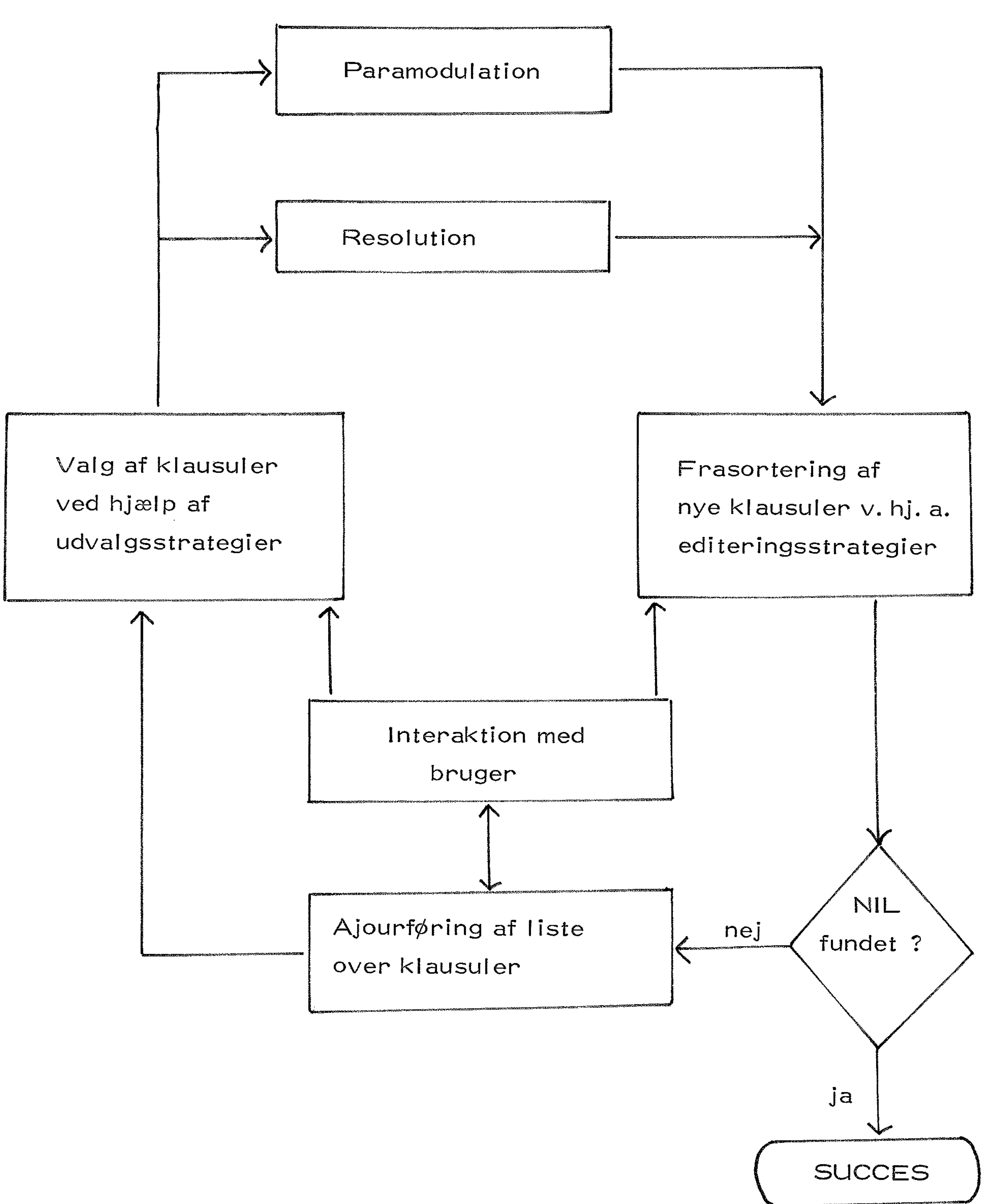
2. Ud fra disse klausuler genereres alle mulige nye klausuler ved resolution og/eller paramodulation.

3. Ved hjælp af de specificerede editeringsstrategier frasorteres evt. nogle af de nye klausuler.

4. Hvis ikke en modstrid (NIL) er fundet, og teoremet dermed bevist, lagres de nye klausuler, der har passeret editeringen.

Efter hvert gennemløb af denne løkke har brugeren mulighed for at afbryde beregningen og gennem interaktive kommandoer opnå flg.:

1. listning på dataskærmen af de hidtil lagrede klausuler.
2. listning af de klausuler, der har været aktive ved deduktionen af en bestemt klausul.



figur 1.

3. beregning og listning af samtlige resolventer og/eller paramodulanter, der kan dannes ud fra to valgte klausuler.

4. tilføjelse eller sletning af klausuler på den lagrede liste.

5. ændring af specifikationen af de ønskede udvalgs- og editeringsstrategier.

Udvalgsstrategier

Blandt de udvalgsstrategier, der står til brugerens rådighed, er dels de tidligere beskrevne Set of Support og Unit Preference og dels 3 andre af de i litteraturen hyppigst forekommende strategier, nemlig:

Resolution med hensyn til en model (R_1).
Resolution with merging (R_2).
Ancestry Filter Form (R_3).

Disse tre strategier er udførligt beskrevet i Luckham [3] og er alle af samme type som Set of Support, dvs. de tillader kun resolutioner ud fra L og M, hvis disse opfylder en til strategien hørende betingelse. I [3] er bevist, at både R_1, R_2 og R_3 er fuldstændige, og yderligere er der vist en række resultater vedr. fuldstændigheden af de mere restriktive strategier, man får ved at benytte en kombination af to af de nævnte strategier. Således bevares fuldstændigheden, hvis R_2 eller R_3 anvendes sammen med Set of Support, og ligeledes hvis R_2 og R_3 anvendes sammen, mens både kombinationen af R_1 med R_2 og R_1 med R_3 er ufuldstændige.

Med det formål at begrænse antallet af paramodulanter har brugeren endvidere mulighed for at specificere en liste $\mathcal{L}$ af lighedsprædikater

$$\mathcal{L} = \{ t_1 = s_1, t_2 = s_2, \ldots, t_n = s_n \},$$

ordnet så at s_i aldrig er mere kompleks end t_i. Denne liste udnyttes på den måde, at det ved paramodulation ind i en klausul M undersøges, om der i M forekommer en variant af et t_i fra $\mathcal{L}$, der kan udskiftes med den tilsvarende variant af s_i. Hvis dette er tilfældet, undersøges det igen, om den modificerede klausul M indeholder en variant af et t_j, der kan udskiftes med den tilsvarende variant af s_j. Dette gentages, så længe det er muligt, og det er så kun den sidstfrembragte variant af M, der videregives til editeringsfasen.

<u>Editeringsstrategier</u>

Brugerens mulighed for at bestemme den editeringsstrategi, der skal anvendes, består i det væsentlige i, at han har rådighed over følgende parametre:

1: En grænse for antallet af sammensætninger af funktionssymboler. Alle klausuler, hvori denne grænse overskrides, bliver forkastet.

2: En grænse på længden (dvs. antal literals) af en klausul.

3: Elimination af trivielle deduktioner. Dette sker ved at specificere en liste $\mathcal{M}$ af klausuler med kun ét literal (f. eks. en delmængde af startklausulerne). Hvis en ny klausul har formen $\neg l_1 \vee l_2$ og der findes en substitution Θ så $\Theta(l_1) \in \mathcal{M}$, og der findes en yderligere substitution Ψ så $\Psi(\Theta(l_2)) \in \mathcal{M}$, så elimineres den pågældende klausul. Grunden til, at brugeren selv kan afgøre, om og i hvor høj grad han ønsker sådanne klausuler fjernet, er, at man ved denne elimination ikke længere har garanti for at en evt. fuldstændighed af en udvalgsstrategi bevares.

<u>Erfaringer med systemet</u>

Gennem en lang række experimenter med det her beskrevne system har forfatterne indhøstet erfaringer om virkningen af forskellige kombinationer af udvalgs- og editeringsstrategier, og som man kunne vente, tegner billedet sig ret broget. Ingen strategi har vist sig overlegen over for alle problemtyper, men i de fleste tilfælde har kombinationen af R_1 med R_3, skønt ufuldstændig, vist sig yderst effektiv, mens f. eks. Unit Preference ofte gør mere skade end gavn, idet den giver anledning til for mange unyttige klausuler i starten af søgningen. Derimod kan Unit Preference undertiden være nyttig, hvis brugeren på et tidspunkt under søgningen er blevet utålmodig og ønsker at undersøge, om overgang til denne strategi kan forcere et hurtigt bevis frem.

I almindelighed starter man med en ufuldstændig men erfaringsmæssig effektiv kombination af strategier. Hvis dette ikke giver resultat, kan man så evt. udelade nogle af de parametre, der forårsager ufuldstændigheden, og prøve igen, og på dette punkt har man naturligvis

stor glæde af de tidligere nævnte teoretiske fuldstændighedsresultater. Et forsøg på at bevise et teorem kan mislykkes af forskellige årsager. Hvis det skyldes, at lagerkapaciteten for hurtigt bliver fyldt med klausuler, kan man forsøge enten med en mere restriktiv udvalgsstrategi eller med snævrere grænser ved editeringen. Hvis det derimod er for stort tidsforbrug, der er problemet, og den dybde i søgningen, der nås, er væsentlig større end forventet, kan man forsøge med en mindre restriktiv udvalgsstrategi eller videre grænser ved editeringen. Dersom det meste af tiden viser sig at blive brugt til editering, kan det omvendt tyde på, at man bør vælge en mere restriktiv udvalgsstrategi.

Det er således et meget fleksibelt system, der står til brugerens disposition, og en effektiv udnyttelse af systemet kræver et grundigt kendskab til samspillet mellem de forskellige strategiingredienser, et kendskab der kun kan opnås gennem praktiske erfaringer med systemet. Det er derfor meget vanskeligt at angive et mål for systemets kapacitet, men forfatterne angiver som eksempel på dets formåen, at man ved hjælp af programmet har fundet beviser for nogle teoremer inden for ternary boolean algebra, som er publiceret uden beviser i Notices of the American Mathematical Society, 1969.

AFSNIT 3

TRÆMETODEN

Efter i de to foregående afsnit at have givet en summarisk oversigt over arten af de arbejder, der er gjort inden for theorem proving v.hj.a. resolutionsmetoden, vil jeg i dette afsnit give en beskrivelse af den metode, som jeg har benyttet i det theorem proving system, der er dokumenteret i næste afsnit. Det er den såkaldte træmetode, som er udviklet og beskrevet af R.C. Jeffrey i hans lærebog i elementær logik [7].

Først skal imidlertid gives en præcis definition af syntaxen for 1. ordens prædikatkalkylen med den notation, der med enkelte modifikationer er benyttet i næste afsnit.

Prædikatkalkylen

Prædikatkalkylens alfabet består af flg. primitive symboler:

Konstanter	: c1, c2, c3, ...
Variable	: x1, x2, x3, ...
Prædikatsymboler	: P_1, P_2, P_3, ...
Lighedsprædikat	: =
Funktionssymboler	: F_1, F_2, F_3, ...
Logiske tegn	: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\forall$, $\exists$.
Hjælpesymboler	: , (komma), [,], (,).

Til hvert prædikatsymbol er der knyttet ét af tallene 0, 1, 2, ... og til hvert funktionssymbol er der knyttet ét af tallene 1, 2, ..., der kaldes prædikatets henholdsvis funktionens <u>orden.</u> Blandt de strenge, der kan konstrueres over dette alfabet, defineres følgende klasser:

<u>Termer</u>:

1: Variable og konstanter er termer

2: Hvis F er et funktionssymbol af orden n ($n \geq 1$), og t1, t2, ... tn er termer så er F[t1, t2, ..., tn] en term.

3: Ingen andre strenge er termer.

En term, hvori der ikke optræder variable, kaldes en <u>konstant term</u>.

Atomare formler:

1: Hvis P er et prædikatsymbol af orden 0, er P en atomar formel.

2: Hvis P er et prædikatsymbol af orden n ($n \geq 1$), og t1, t2, ..., tn er termer, så er P[t1, t2, ..., tn] en atomar formel.

3: Hvis t1 og t2 er termer, så er t1 = t2 en atomar formel.

4: Ingen andre strenge er atomare formler.

Velformede formler:

1: En atomar formel er en velformet formel.

2: Hvis A og B er velformede formler, og x er en variabel, så er (A), $\neg A$, $\forall x A$, $\exists x A$, $(A \vee B)$, $(A \wedge B)$ og $(A \rightarrow B)$ velformede formler.

3: Ingen andre strenge er velformede formler.

I de velformede formler $\forall\, x\, A$ og $\exists\, x\, A$ siges strengen A at være scope for henholdsvis $\forall\, x$ eller $\exists\, x$. En forekomst af en variabel x i en velformet formel siges at være bundet, netop hvis den forekommer inden for scopet af en alkvantor $\forall\, x$ eller en existenskvantor $\exists\, x$, eller hvis forekomsten er en del af strengen $\forall\, x$ eller $\exists\, x$. Ellers siges forekomsten at være fri. En sætning er en velformet formel uden frie variable.

Konventioner:

I en sætning kan udelades eventuelle yderste parantessæt.

En formel skal i det følgende altid betyde en velformet formel.

Formler af typen $\neg(t1 = t2)$ skrives også $t1 \neq t2$.

Hvis der i en sætning forekommer flere kvantorer, skal de tilhørende variabelnavne vælges forskelligt.

Derimod indføres der ingen konventioner ang. en prioritering mellem de logiske tegn $\wedge$, $\vee$, og $\rightarrow$.

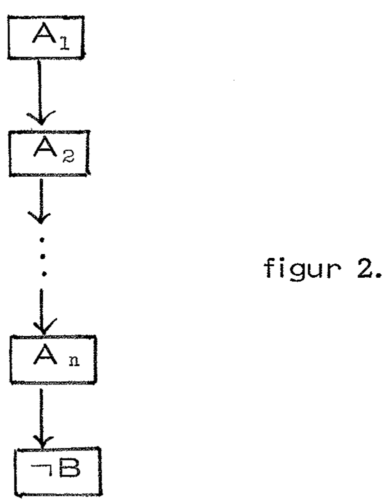
Eksempler på sætninger:

1: $P_1\,[c1,\ F_1[c2]] \rightarrow \forall x1\ \ (x1 = F_1\,[c1])$.

2: $\forall\, x\, 1\ \neg \exists\, x\, 2\, (P_1\,[x1,\ c2] \wedge (P_2\,[x2] \vee \neg P_2\,[c1]))$.

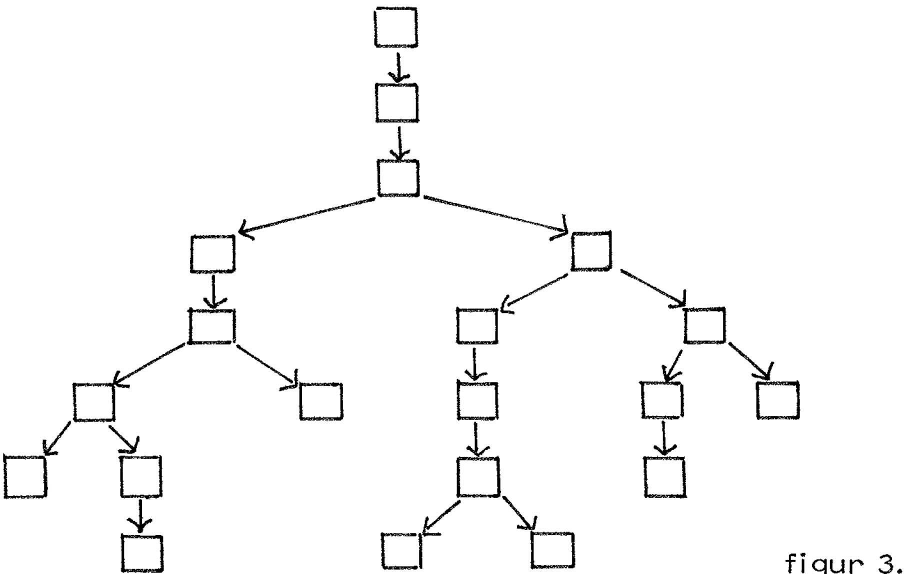
Inferensregler

Ved træmetoden bevises et teorem af formen $A_1, A_2, \ldots, A_n \vdash B$ ligesom ved resolutionsmetoden ved at vise, at mængden $\{A_1, A_2, \ldots, A_n, \neg B\}$ er unsatisfiable. Hver af disse startsætninger associeres med



figur 2.

en knude i en orienteret graf, som figur 2 viser. Metoden består nu i, at man udvider denne graf nedad ved at tilføje nye sætninger (knuder), som kan udledes ud fra de sætninger, der allerede er i grafen, v.hj.a. en række forskellige inferensregler. Nogle af disse regler vil medføre, at to alternative sætninger skal tilføjes grafen, som derfor vil få en struktur som et træ med sætningen A_1 (evt. $\neg B$ hvis mgd. af præmisser er tom) som topknude (rod). (figur 3).



figur 3.

Ved et blad vil vi i det følgende forstå en knude uden efterfølgere, og en sti skal betyde mængden af de sætninger, der ligger på en orienteret vej gennem grafen fra topknuden til et blad.

Ved anvendelsen af en inferensregel på en sætning i træet vil der til hver sti, som den pågældende sætning tilhører, blive føjet en ny struktur bestående af en eller flere sætninger. Til gengæld vil i de fleste tilfælde den sætning, der bliver udviklet v.hj. a. inferensreglen, kunne slettes fra træet. På de kommende illustrationer vil den til sætningen hørende knude dog blive bevaret i træstrukturen, og i stedet vil det blive markeret med en *, at den pågældende sætning kan betragtes som slettet. Herved bevares den binære træstruktur.

Hvilken inferensregel, der kan anvendes på en given sætning, afhænger af sætningens form. Alle inferensreglerne med undtagelse af 2 er angivet skematisk på figur 4.

For alle inferensreglerne i figur 4 gælder det, at den sætning, der udvikles v.hj.a. reglen, kan slettes fra træet. Dette gælder ikke for de to resterende inferensregler:

Regel for alkvantor:

Hvis der i en sti forekommer en sætning af formen $\forall x A$, og der i stien netop forekommer flg. konstante termer t1, t2, ... tn, kan der til stiens blad føjes en lineær struktur bestående af de af sætningerne A^x_{t1}, A^x_{t2}, ..., A^x_{tn}, der ikke tidligere er tilføjet stien. (A^x_{ti} betegner den sætning, der fås ved at erstatte alle forekomster af x i A med ti). Såfremt der ikke forekommer nogen konstante termer i stien, vælges en ny konstant c, og sætningen A^x_c føjes til stien.

Formen af den sætning, hvorpå inferensreglen kan anvendes.	Den nye struktur, der skal tilføjes alle blade under den sætning, som udvikles v. hj. a. inferensreglen.
$\neg\neg A$ (negneg)	↓ □ A
$A \vee B$ (or)	A□ ↙ ↘ □B
$\neg(A \vee B)$ (nor)	↓ □ $\neg A$ ↓ □ $\neg B$
$A \wedge B$ (and)	↓ □ A ↓ □ B
$\neg(A \wedge B)$ (nand)	$\neg A$□ ↙ ↘ □ $\neg B$
$A \rightarrow B$ (impl)	$\neg A$□ ↙ ↘ □ B
$\neg(A \rightarrow B)$ (nimpl)	↓ □ A ↓ □ $\neg B$
$\neg\forall x A$ (negall)	↓ □ $\exists x \neg A$
$\neg\exists x A$ (negex)	↓ □ $\forall x \neg A$
$\exists x A$ (ex)	□ A^{x}_{c} der fremkommer ved at erstatte alle forekomster af x i A med en ny konstant c , der ikke forekommer i stien.

figur 4. Inferensregler.

Regel for lighedstegn:

Hvis der i en sti forekommer en sætning af formen t1 = t2 eller t2 = t1 samt en sætning A, hvori termen t1 indgår, kan der til denne sti føjes den sætning, der fås ved at erstatte en forekomst af t1 i A med t2, medmindre denne nye sætning allerede tidligere er tilføjet stien.

Af inferensreglernes udformning følger, at der efter et vilkårligt antal anvendelser af disse gælder, at såfremt der findes en interpretation, der tilfredsstiller mindst én af træets stier, vil der også findes en interpretation, der tilfredsstiller alle startsætningerne, og omvendt. At vise, at mængden af startsætninger er unsatisfiable, kan derfor ske ved at udvikle træet så langt, at alle stierne kan lukkes, idet en sti lukkes, når det fremgår, at den ikke kan tilfredsstilles af nogen interpretation. Dette kan ske på to måder:

1. Hvis en sti indeholder en sætning og dens negation, kan stien lukkes.

2. Hvis en sti indeholder en sætning af formen $t \neq t$, kan stien lukkes.

Når en sti er lukket betyder det altså, at man ikke behøver at udvikle træet videreunder den pågældende stis blad, og at man derfor i den videre udvikling af træet kan se bort fra de sætninger, som kun forekommer i denne sti (en <u>gren</u> af træet). På de følgende illustrationer er det markeret med et X under et blad, hvis den pågældende sti er lukket.

Eksempler.

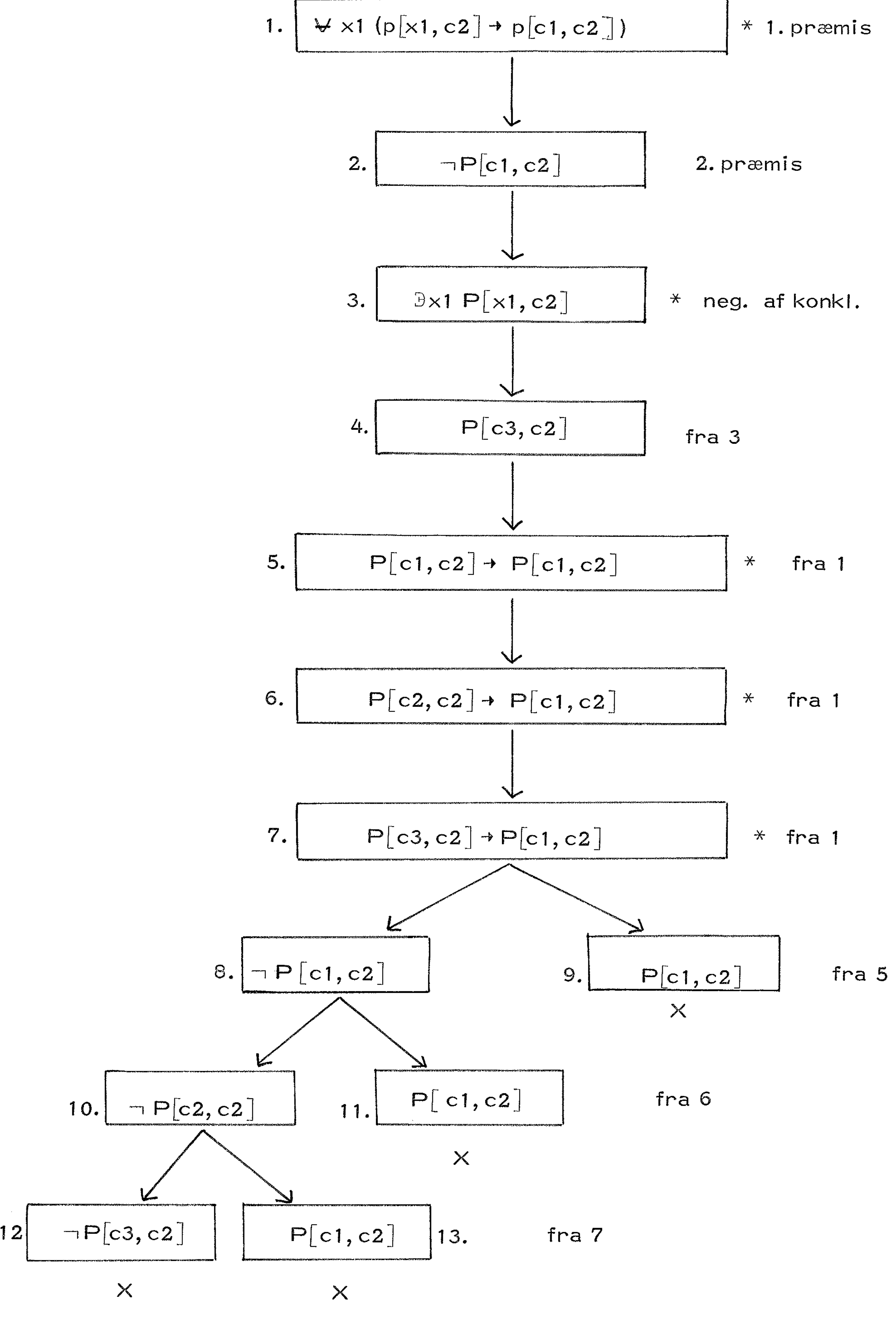
Som illustration af inferensreglernes anvendelse er der på de følgende figurer vist detaljerede eksempler på, hvordan beviserne for to meget simple theoremer kan forløbe v. hj. a. træmetoden.

I figur 5 er bevist, at sætningen $\neg \exists x1 \; P[x1, c2]$ følger logisk af præmisserne $\forall x1 \; (P[x1, c2] \rightarrow P[c1, c2])$ og $\neg P[c1, c2]$.

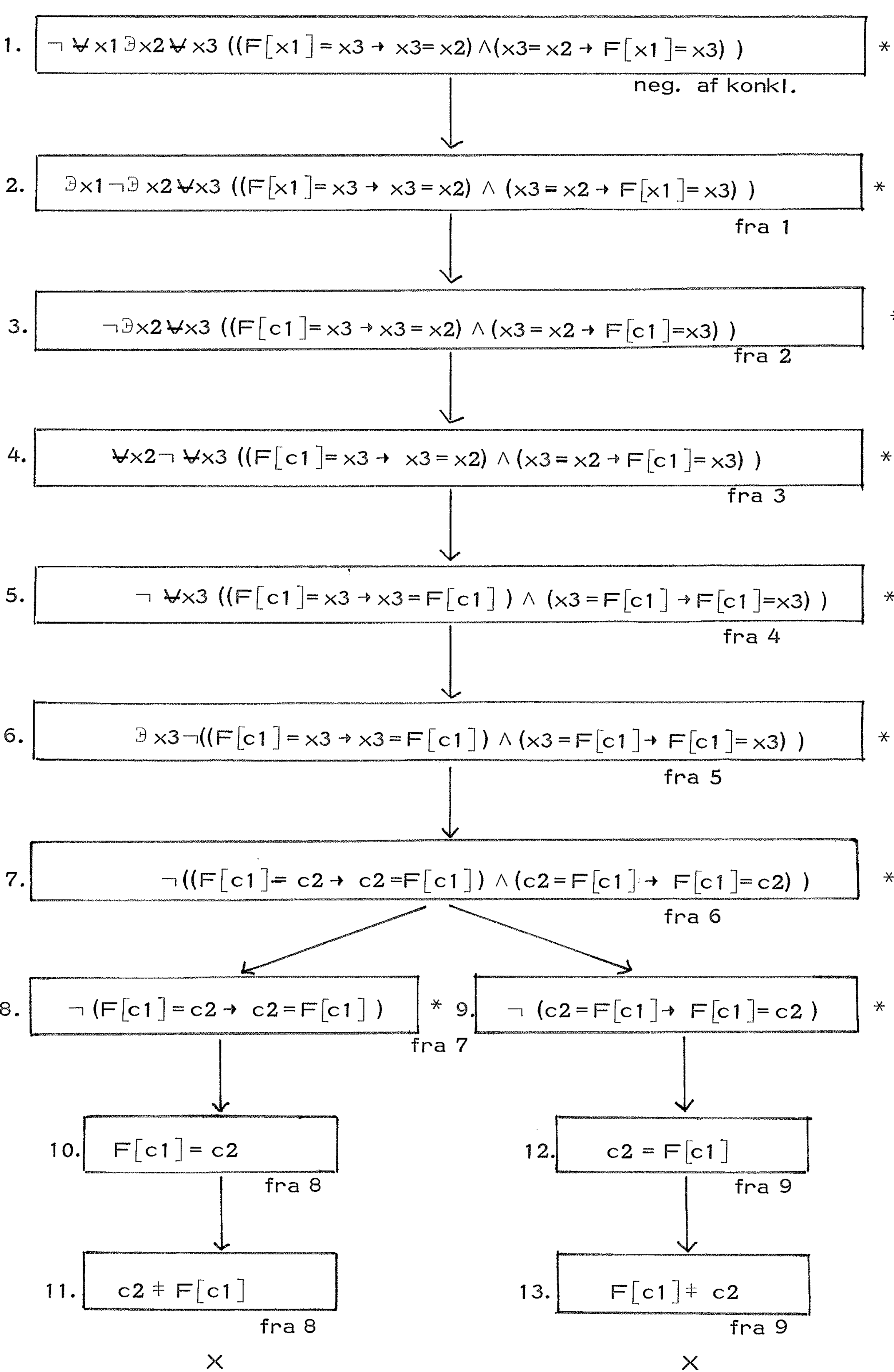
I figur 6 er bevist, at sætningen

$\forall x1 \; \exists x2 \; \forall x3 \; ((F[x1] = x3 \rightarrow x3 = x2) \wedge (x3 = x2 \rightarrow F[x1] = x3))$ er gyldig,

hvilket retfærdiggør vores brug af funktionssymbolerne.



figur 5.



figur 6.

Til sammenligning er det i afsnit 5 vist, hvorledes de maskinelle beviser for de samme sætninger forløber.

Strategier.

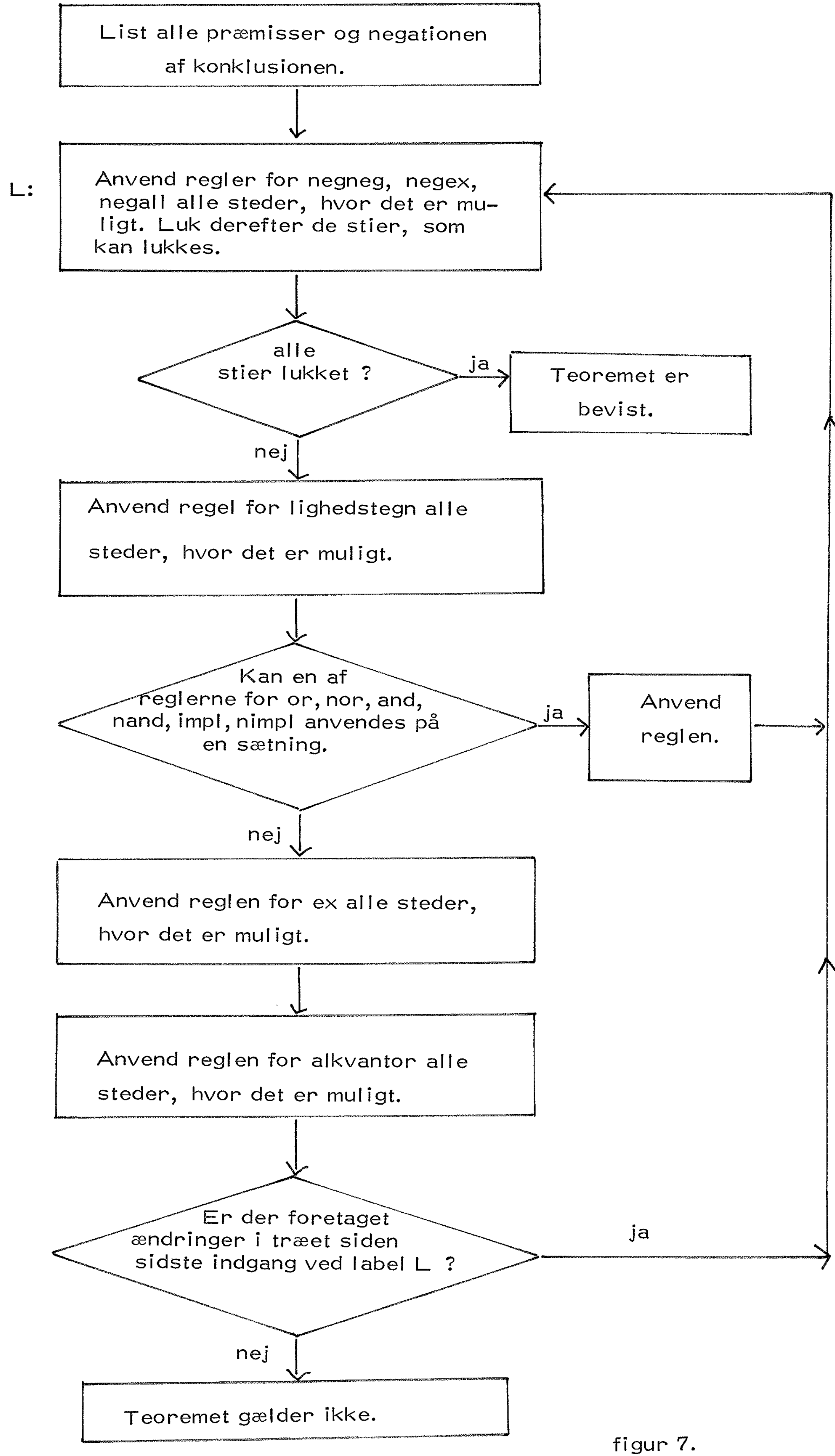
Afgørende for forløbet af et forsøg på at bevise et teorem er naturligvis den rækkefølge, i hvilken man vælger at udvikle sætningerne. I Jeffreys bog [7] er vist, at den algoritme, der er beskrevet ved hjælp af rutediagrammet i figur 7, er et eksempel på en fuldstændig strategi, idet man ved anvendelse af algoritmen på en unsatisfiable mængde af startformler altid vil få alle stier lukket i løbet af endelig tid. Hvis mængden af startformler er satisfiable vil der derimod indtræffe ét af flg. tilfælde:

1. på et vist tidspunkt kan der ikke genereres flere nye sætninger v. hj. a. inferensreglerne, og der er stadig åbne stier i træet.

2: algoritmen stopper aldrig.

Denne fuldstændighed er imidlertid af mindre interesse, da en slavisk udførelse af algoritmen oftest vil være uigennemførlig på grund af det store antal sætninger, som først og fremmest reglen for alkvantor og lighedstegn vil give anledning til. Derfor er der i det foreliggende program ikke indbygget nogen "færdig" strategi, men ideen er, at brugeren selv ved interaktion med programmet skal dirigere udviklingen af træet. Dette sker dels gennem en række kommandoer, der forårsager et enkelt skridt i opbygningen af træet, og dels gennem muligheden for at definere en delstrategi, der bevirker en sekvens af sådanne skridt. Disse elementer kan så indgå som byggestene i den overordnede strategi, som brugeren ønsker at følge, og på den måde bliver den omtalte fuldstændige algoritme kun én blandt utallige andre, som brugeren kan experimentere med.

Jeffrey diskuterer i sin bog [7] overhovedet ikke problemerne vedrørende en evt. implementering af et theorem proving system på en datamat, og så vidt jeg ved, er der hidtil kun blevet gjort et enkelt forsøg på at basere et sådant system på træmetoden, nemlig ved Brandeis Uni-



figur 7.

versity, hvor Cohen & Rubin har implementeret et system, der er beskrevet i [8]. Det er beskrivelsen af dette system, der har givet inspiration til det foreliggende arbejde, først og fremmest gennem påvisningen af metodens egnethed til interaktive anvendelser. Det pascalprogram, der skal beskrives i næste afsnit, betegner en udvidelse i forhold til Cohen & Rubins algolsystem, der ikke omfatter funktioner, lighedstegn eller mulighed for definition af delstrategier, ligesom de to systemer adskiller sig væsentligt fra hinanden med hensyn til datarepræsentation, lagerallokering og hele programmeringsmæssige design.

AFSNIT 4

DOKUMENTATION AF PROGRAM

Dette afsnit består af en dokumentation af det theorem proving system, som jeg har implementeret på RECAU's CDC 6400 anlæg. Programmet er skrevet i Pascal [9] og er designet til interaktiv kørsel via en display terminal. Dokumentationen vil bestå dels af en beskrivelse af, hvordan systemet præsenterer sig for en bruger, og dels af en gennemgang af den interne datarepræsentation og de vigtigste dele af programmets virkemåde. For programmeringsmæssige enkeltheder henvises til Appendix A (programudskrift), og for mere udførlige detaljer vedr. anvendelsen af systemet henvises til Appendix B (brugervejledning).

Ved kommunikationen mellem program og bruger benyttes den syntax for prædikatkalkylen, som blev defineret i afsnit 3, med følgende ændringer i notationen afført af display terminalens karaktersæt:

1) $\forall$ (alkvantor) skrives A
2) $\exists$ (existenskvantor) skrives
3) $\neg$ (negation) skrives - (minustegn)
4) $\rightarrow$ (implikation) skrives $\vdash$→

Endvidere er indført følgende begrænsninger:

1) Tilladte prædikatsymboler: P, Q, R, S, T, U, V.
2) Tilladte funktionssymboler: F, G, H, I, J, K, L, M, N.
3) Tilladte konstanter: c1, c2, ..., c63.
4) Tilladte variable: x1, x2, ..., x63.
5) Funktioner og prædikatsymboler må højst have 3 argumenter.
6) Det højeste antal sætninger i træet : 2048.

Begrundelsen for disse begrænsninger fremgår af den valgte datarepræsentation og er naturligvis et kompromis mellem generalitet og programmeringseffektivitet.

For brugeren, der ønsker at bevise et teorem, fremtræder systemet på den måde, at han først bedes om at indtaste startsætningerne (dvs. præmisserne og negationen af konklusionen). Derefter foregår konversationen i en løkke, hvor hvert gennemløb består af flg. 3 faser:

1: Programmet skriver på skærmen
TAST NY ORDRE
↱ . ↱

2: På linien, der begynder med ↱ ↱ , skriver brugeren et kommandoord efterfulgt af evt. parametre.

3: Programmet beder evt. om yderligere parametre og udfører den ønskede ordre, hvis denne er lovlig; ellers fås en fejludskrift. Under udførelsen gives på skærmen information om resultatet af den udførte ordre, herunder bl. a. hvilke nye sætninger, der tilføjes træet, hvilke stier, der lukkes, og meddelelsen TEOREMET ER BEVIST, hvis det lykkes at lukke alle stier i træet.

Hver sætning, som indgår i træstrukturen, tildeles af programmet et nummer, der tjener til identifikation af de pågældende sætninger under dialogen mellem bruger og program. Tilsvarende er hver sti i træet karakteriseret ved et nummer. Udviklingen af træet v. hj. a. reglerne i foregående afsnit kan ske på to principielt forskellige måder:

1: Brugeren kan vælge at arbejde på hele træet. Dette svarer nøje til beskrivelsen af træmetoden i foregående afsnit, dvs. hvis en sætning udvikles v. hj. a. en inferensregel, føjes den nye sætningsstruktur til alle de stier, som den pågældende sætning tilhører, og den udviklede sætning kan evt. slettes fra træet.

2: Brugeren kan vælge at arbejde kun på en enkelt sti. Hvis en sætning i denne sti udvikles v. hj. a. en inferensregel, føjes den nye sætningsstruktur kun til den pågældende stis blad, og til gengæld kan den udviklede sætning ikke slettes fra træet. Dette be-

grænser hastigheden i væksten af træet og kan benyttes, hvis man vil forsøge at lukke en enkelt sti ad gangen.

Brugeren kan under kørslen skifte mellem disse to arbejdsmåder v. hj. a. kommandoen DEL.

Interaktive kommandoer

I det følgende omtales alle de interaktive kommandoer, som står til rådighed for brugeren, idet der henvises til brugervejledningen for detaljer vedr. inputformater, fejludskrifter osv.

DEL n

hvor n er tallet 0 eller nummeret på en sti. Ordren bestemmer hvilken del af træet, der skal arbejdes på indtil næste DEL-kommando. $n = 0$ betyder, at der skal arbejdes på hele træet, og $n \neq 0$ betyder, at der skal arbejdes på den sti, som n angiver. Ved starten af programmet arbejdes der på hele træet indtil første DEL-kommando.

FOCUS n_1 n_2 ... n_k

hvor n_1, n_2, ..., n_k er numre på sætninger af formen $A \vee B$, $\neg(A \vee B)$, $A \wedge B$, $\neg(A \wedge B)$, $A \rightarrow B$, $\neg(A \rightarrow B)$, $\exists x A$, $\neg \exists x A$ eller $\neg \forall x A$. Forudsat sætningerne tilhører den del af træet, der arbejdes på, udvikles disse v. hj. a. reglerne i figur 4, og den tilhørende ændring af træet foretages.

ALKV t_1 t_2 ... t_k

hvor t_1, t_2, ..., t_k er konstante termer. Programmet beder brugeren skrive numrene på en række sætninger, der alle styres af en alkvantor. Hver af disse udvikles (såfremt de tilhører den del af træet, der arbejdes på) v.hj.a. termerne t_1, t_2, ..., t_k og den tilhørende ændring af træet foretages.

SUB V n og SUB H n

hvor n er nummeret på en sætning af formen $t_1 = t_2$. Programmet beder brugeren skrive numrene på en vilkårlig række af sætninger i den del af træet, der arbejdes på. Disse sætninger

udvikles v.hj.a. reglen for lighedstegn, således at de nye sætninger, der tilføjes træet, fremkommer ved at udskifte alle forekomster af t_1 med t_2, såfremt kommandoordet er SUB V, mens den modsatte substitution foretages, såfremt kommandoordet er SUB H.

NB: Hvis man i en sætning kun ønsker en enkelt forekomst af t_1 eller t_2 udskiftet, kan man bruge kommandoen TILFØJ.

SKRIV n

hvor n er nummeret på en sætning i den del af træet, der arbejdes på. Den pågældende sætning udskrives.

STI n

hvor n er nummeret på en sti. Der udskrives en liste over alle ikke-slettede sætninger i stien med sætningernes nr. og udseende.

TRÆ

Der udskrives en liste over alle åbne stier i træet med angivelse af numrene på de ikke-slettede sætninger, som ligger i de forskellige stier. Endvidere udskrives der oplysning om hvor stor en del af den disponible lagerkapacitet, der er beslaglagt.

TILFØJ s

hvor s er en sætning i prædikatkalkylen. Sætningen tilføjes alle stierne i træet eller kun en enkelt sti afhængig af sidste DEL-kommando. Denne ordre kan f. eks. anvendes, hvis brugeren ønsker at tilføje nogle extra præmisser i forsøget på at bevise et teorem.

SLET n

hvor n er nummeret på en sætning i den del af træet, der arbejdes på. Den pågældende sætning slettes af træet.

<u>LUK n</u>

hvor n er nummeret på en åben sti. Den pågældende sti vil blive blive lukket. Denne ordre kan f. eks. anvendes, hvis brugeren ud fra sin logiske viden kan se, at en sti indeholder en modstrid, og derfor vil spare de beregninger, der fører til, at programmet opdager denne modstrid, og selv lukker stien.

<u>DEF n</u>

hvor n er et af tallene 1, 2, 3, 4, 5. Denne ordre tillader brugeren at definere en strategi, der lagres under nummeret n, og som ved senere anvendelser af ordren GO kan bringes til udførelse. En strategi fastlægges v. hj. a. fire parametre, som programmet beder brugeren om at specificere én efter 'en. Disse parametre bestemmer hvilke sætninger i træet, der på udførelsestidspunktet skal udvikles, og hvor langt denne udvikling skal fortsættes. De strategier, der defineres ved denne ordre, kan kun anvende inferensreglerne i figur 4, og de kan altså ikke udvikle sætninger v. hj. a. reglen for alkvantor eller lighedstegn. De fire parametre er følgende:

<u>1. FORMLER</u>: Værdien af denne parameter bestemmer hvilke af de sætninger, som befinder sig i træet, når ordren GO gives, der skal udvikles. "Formler" kan tildeles flg. værdier:

<u>Numre</u> : bevirker, at programmet ved ordren GO beder om numrene på de sætninger, der skal udvikles.

<u>Prnavn</u> : bevirker, at programmet ved ordren GO beder brugeren om at indtaste nogle prædikatsymboler. Kun de sætninger i træet, hvori ét af disse navne indgår, udvikles.

<u>Udengren</u> : bevirker, at kun de sætninger udvikles, som ikke giver anledning til dannelsen af nye stier.

<u>Alle</u> : bevirker, at alle sætninger (undtagen alkvantor og lighedstegn) udvikles.

<u>2. VIDERE</u>: Værdien af denne parameter bestemmer, om de nye sætninger, der tilføjes træet under udførelsen af en strategi, skal videreudvikles, hvis dette er muligt. "Videre" kan tildeles flg. værdier:

<u>Nej</u> : bevirker, at de nye sætninger ikke videreudvikles.

<u>Tilgren</u> : bevirker, at de nye sætninger udvikles videre, såfremt dette ikke giver anledning til yderligere forgreninger i træet.

<u>Allenye</u> : bevirker, at alle nye sætninger udvikles "tilbunds", dvs. sålænge inferensreglerne fra figur 4 kan finde anvendelse.

<u>3. NYTUR:</u> Værdien af denne parameter specificeres kun, hvis værdien af "videre" er forskellig fra nej, og bestemmer i hvilken rækkefølge de ved udførelsen af en strategi dannede sætninger skal videreudvikles. "Nytur" kan tildeles flg. værdier:

<u>Først</u> : bevirker, at en sætning fra træet udvikles så langt, som den skal iflg. "videre", inden udviklingen af næste sætning fra det oprindelige træ påbegyndes. Dette svarer til en slags dybde - først udvikling.

<u>Sidst</u> : bevirker, at først udvikles alle de sætninger fra fra træet, der skal udvælges iflg. "formler", ét skridt. De af de nye formler, der skal udvælges iflg. "videre", udvikles dernæst ét skridt, og dette gentages, så længe det er muligt. Dette svarer til en slags bredde - først udvikling.

<u>4. STOPINTERVAL</u>: Værdien af denne parameter giver brugeren mulighed for at stoppe udførelsen af en strategi, hvis denne tager en uheldig retning. Hvis "stopinterval" tildeles værdien n (pos. helt tal), bevirker det, at hver gang n sætninger er blevet udviklet under strategien, spørger programmet, om det skal fortsætte med at afvikle den pågældende strategi. Hvis værdien af "stopinterval" er 0, sker der ingen afbry delse af beregningen.

GO n

hvor n er nummeret på en tidligere defineret strategi. Ordren bringer denne strategi til udførelse efter de retningslinier, der er beskrevet under omtalen af DEF.

VIS n

hvor n er nummeret på en tidligere defineret strategi. Strategiens udseende vises på skærmen ved en listning af de fire parametres værdier. En tidligere defineret strategi kan omdefineres ved fornyet kald af ordren DEF n.

NY

Bevirker, at programmet starter forfra med initialisering og indtastning af nye startsætninger.

BYE

bevirker, at programmet stopper.

Datarepræsentation

Den interne datarepræsentation består af forskellige datastrukturer, der alle er opbygget af en enkelt recordtype, nemlig:

```
elem = packed record
          nr : 1 .. 2048 ;
          navn : char ;
          x, y, z: - 63 .. 63 ;
          back, left, right, formel : pointer ;
          kode : kodetegn
       end ;
```

Her er pointer = ↑ lager ;

lager : class max of elem ;

og kodetegn = (or, nor, and, nand, impl, nimpl, ex, negex, all, negall, p0, np0, p1, np1, f1, p2, np2, f2, lig, ulig, p3, np3, f3).

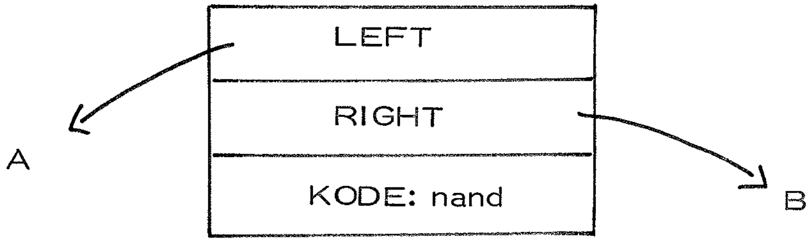
En record af typen elem består altså af flg. felter:

NR
NAVN
X
Y
Z
BACK
LEFT
RIGHT
FORMEL
KODE

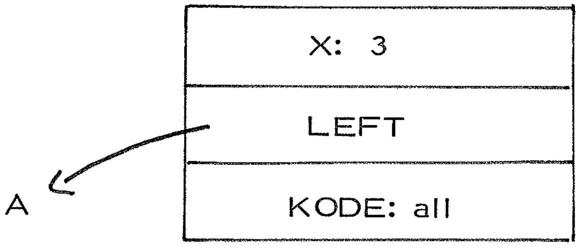
Takket være pascalfaciliteten packed record fylder en sådan record kun 2 ord (á 60 bit) i lageret. Anvendelsen af de forskellige felter i en record afhænger af på hvilken måde, den pågældende record indgår i en datastruktur, og der skal nu gøres rede for opbygningen af disse forskellige strukturer. På de følgende illustrationer er der kun med taget de recordfelter, der har betydning i den pågældende struktur, mens de felter, hvis indhold er udefineret, er udeladt for overskuelighedens skyld.

Repræsentation af formler

Formler af formen $A \vee B$, $\neg(A \vee B)$, $A \wedge B$, $\neg(A \wedge B)$, $A \rightarrow B$ eller $\neg(A \rightarrow B)$ repræsenteres af en record, hvor left er en pointer til formlen A, right er en pointer til formlen B, og kode har værdien henholdsvis or, nor, and, nand, impl eller nimpl. Formlem $\neg(A \wedge B)$ repræsenteres således:



Formler af formen $\exists$ xi A, $\neg\exists$ xi A, $\forall$ xi A eller $\neg\forall$ xi A repræsenteres af en record, hvor left er en pointer til formlen A, feltet x har værdien i og kode har værdien henholdsvis ex, negex, all eller negall. Formlen $\forall$ x3 A repræsenteres således:



Repræsentationen af funktions- og prædikatsymboler er knap så enkel. Her gælder, at funktionens/prædikatets navn repræsenteres som en char-værdi i feltet navn. Betydningen af feltet kode er givet ved:

p0, p1, p2, p3 repræsenterer prædikat med 0, 1, 2, 3 argumenter.
f1, f2, f3 repræsenterer funktion med 1, 2, 3 argumenter.
np0, np1, np2, np3 repræsenterer negation af prædikat med 0, 1, 2, 3 argumenter.
lig repræsenterer en formel af formen $t_1 = t_2$.
ulig repræsenterer en formel af formen $t_1 \neq t_2$.

Første argument repræsenteres af felterne x og left på flg. måde:

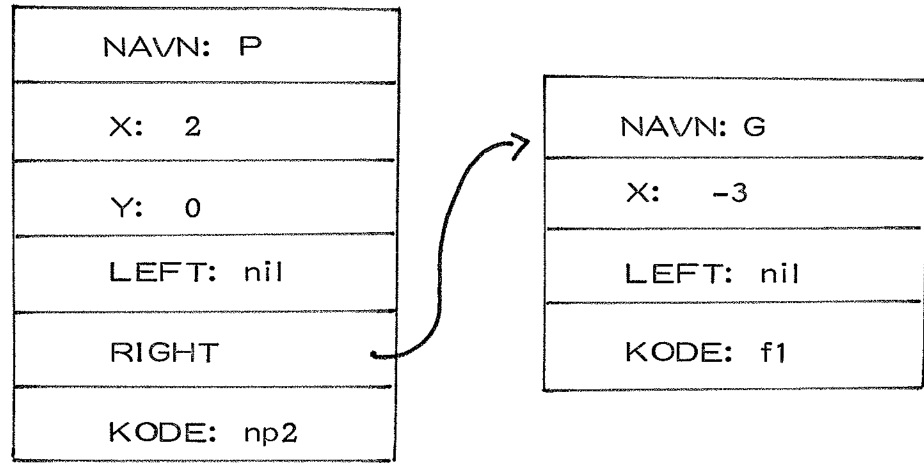
Hvis argumentet er konstanten ci er værdien af feltet x = tallet ÷ i og left = NIL.

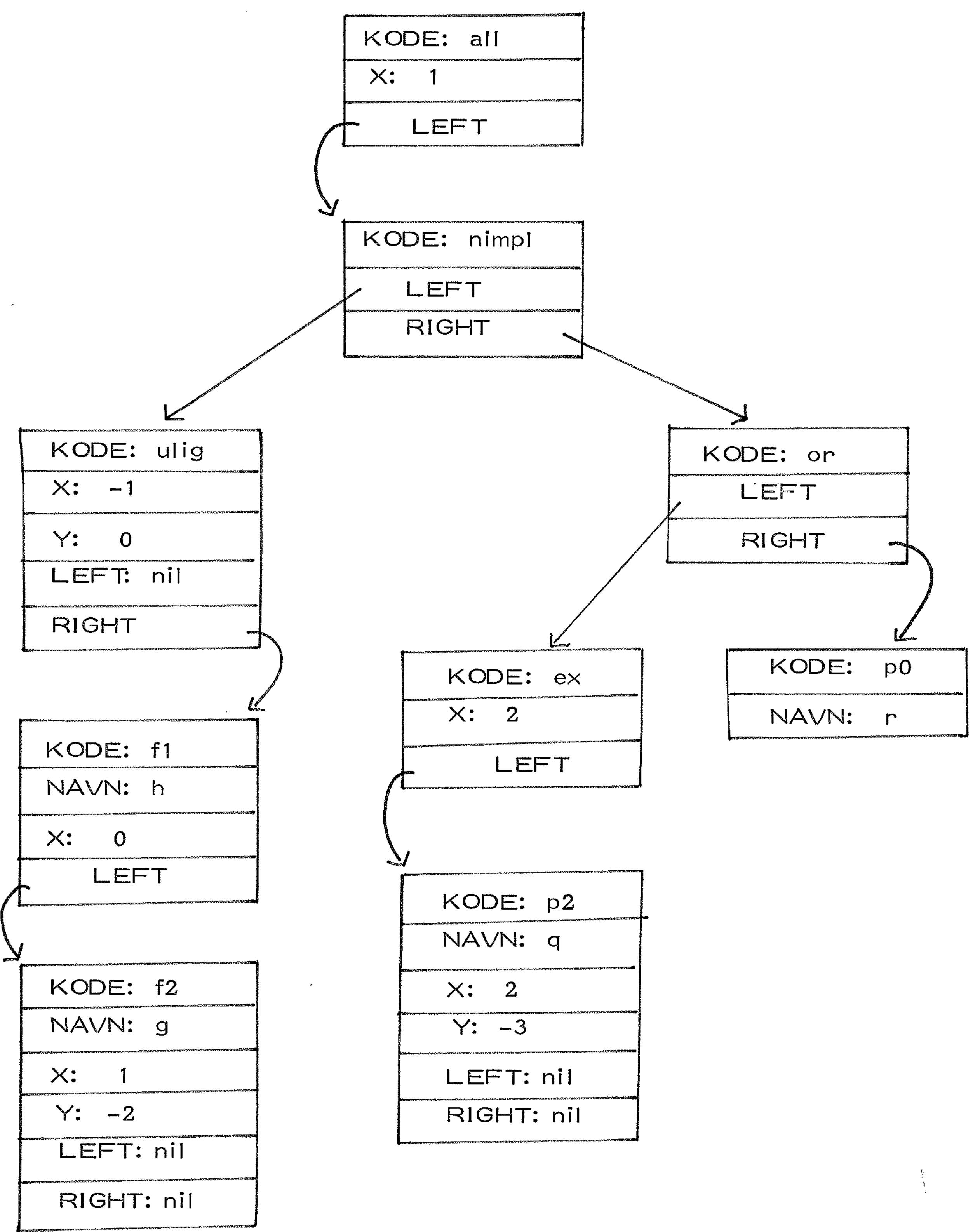
Hvis argumentet er variablen xi er værdien af feltet x = tallet + i og left = NIL.

Hvis argumentet er en funktion er værdien af feltet x = tallet 0, og left er en pointer til den record, som repræsenterer funktionen.

På analog måde repræsenteres 2. argument v. hj. a. felterne y og right og 3. argument v. hj. a. felterne z og back.

Formlen $\neg$ P [x2, G [c3]] repræsenteres således:





figur 8.

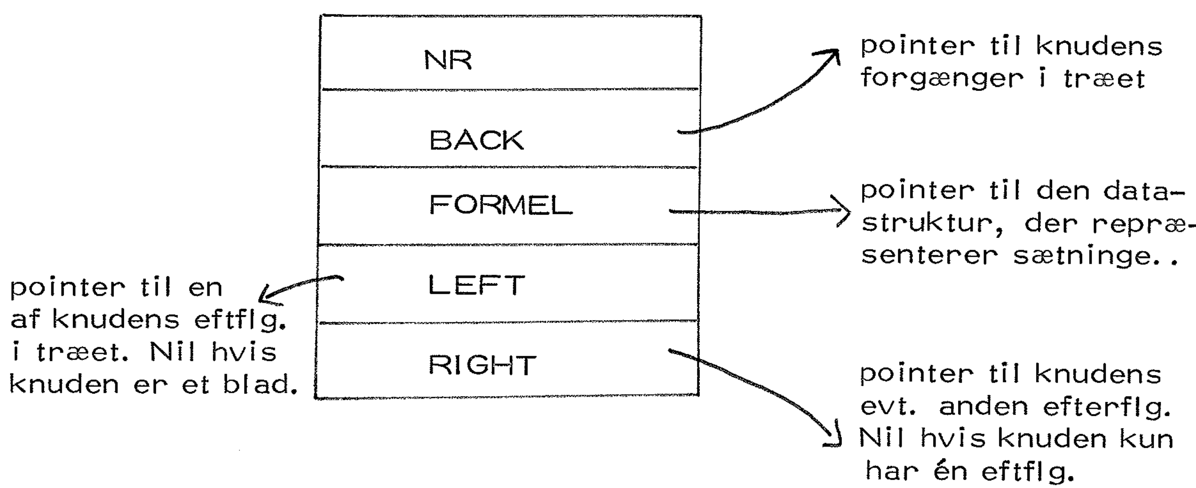
Repræsentation af sætningen: $\forall x1 \neg (c1 \neq H[G[x1, c2]] \rightarrow (\exists x2 Q[x2, c3] \vee R))$.

På figur 8 er som eksempel vist repræsentationen af sætningen:

$\forall x1 \neg (c1 \neq H[G[x1, c2]] \rightarrow (\exists x2\ Q[x2, c3] \vee R))$.

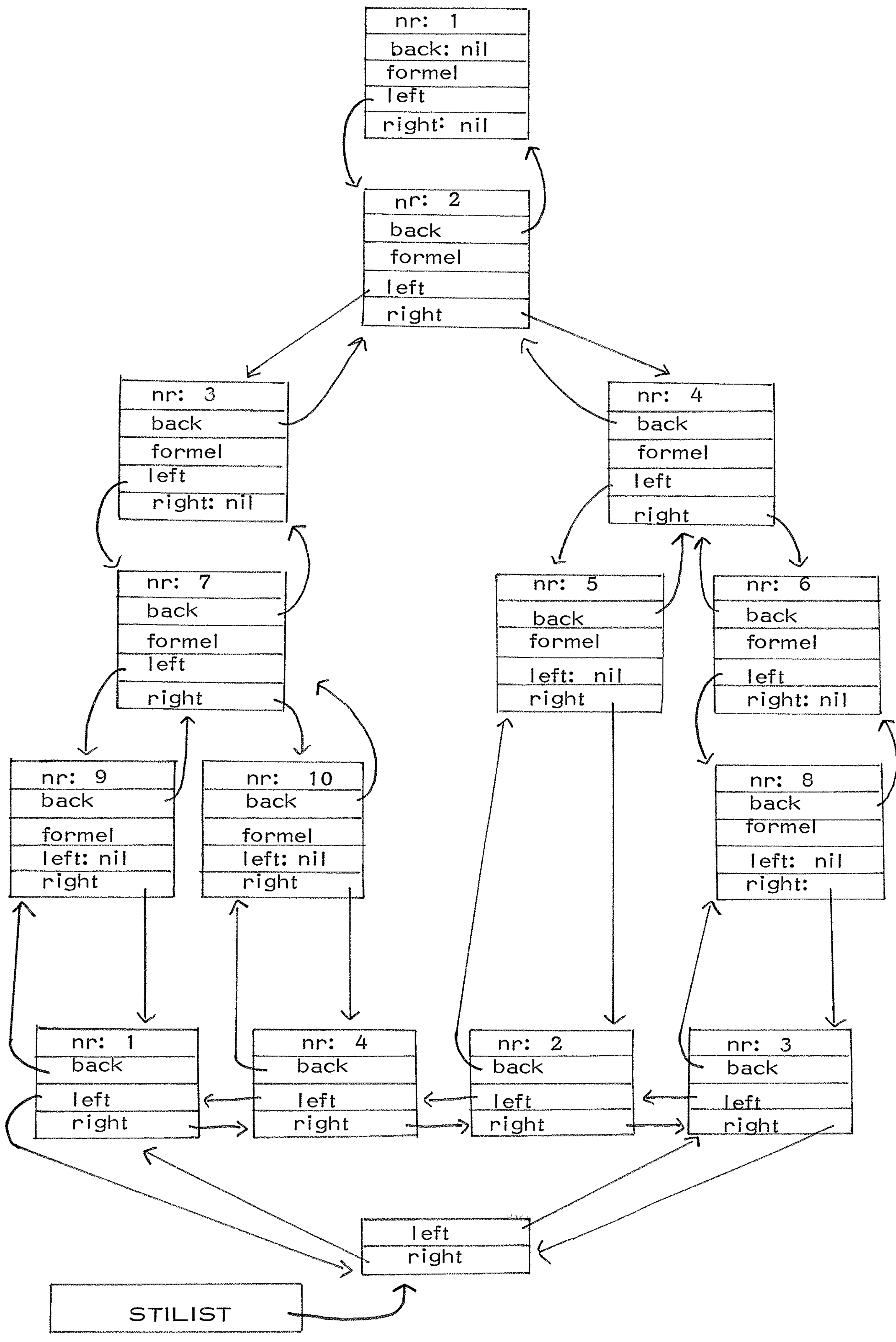
Repræsentation af træ

Den træstruktur, som opbygges under kørsel med programmet, repræsenteres ved, at der til hver sætning hører en knude i træet, repræsenteret ved en record, hvori feltet formel indeholder en pointer til den recordstruktur, som repræsenterer sætningen, og felterne back, left og right indeholder referencer til knudens forgænger og efterfølger(e) i træet:

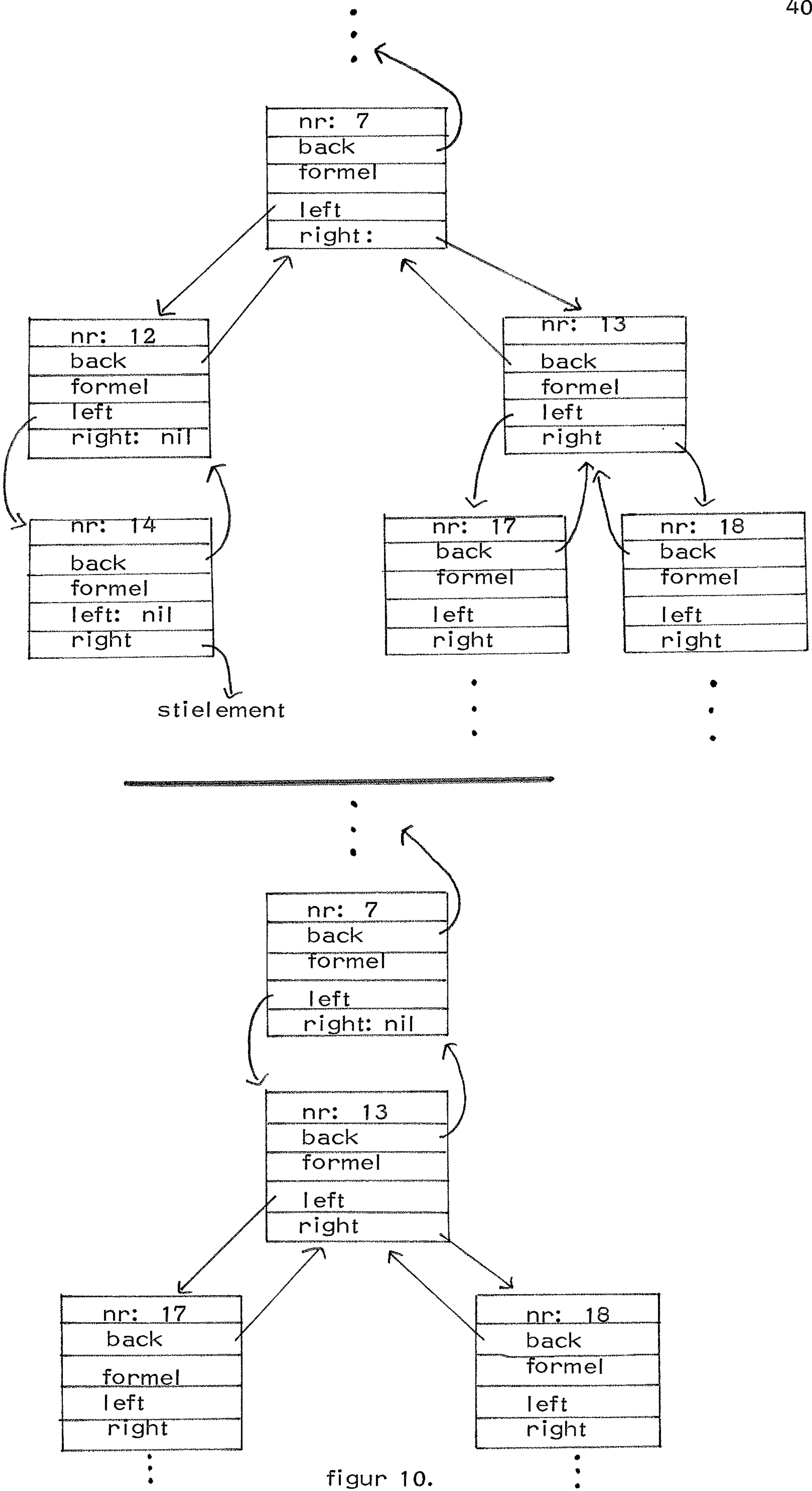


Repræsentationen af et træ er vist i figur 9, hvor der også er med medtaget den stiliste, der beskrives senere.

Hvis en formel kan slettes efter anvendelsen af en inferensregel bevares sætningsknuden i træet, men feltet formel får værdien NIL, og de records der repræsenterer sætningen frigives.(Proceduren SLETFORM).



figur 9.



figur 10.

En del af sætningstræet før og efter stien til venstre lukkes.

Hvis en sti kan lukkes, og alle sætninger i den pågældende gren derfor kan slettes, fjernes imidlertid også de til sætningerne hørende knuder i træet, så at træet til enhver tid kun indeholder åbne åbne stier og beslaglægger færrest mulige records. (proceduren SLETGREN, figur 10)

Stiliste

For hver sti i træet findes et stielement, dvs. en record, hvor feltet nr indeholder det tal, som identificerer stien, og feltet back indeholder en pointer til den pågældende stis blad. Disse stielementer er v. hj. a. felterne left og right organiseret til en cirkulær liste, hvis hoved er en speciel record, som den globale variable STILIST refererer til (figur 9). I stiens blad, hvor feltet left har værdien NIL for at markere, at knuden ingen efterfølgere har, indeholder feltet right en pointer til stielementet, så at man både kan komme fra stiens sætninger til stielementet, hvor information om stiens nr. findes, og den anden vej. Når en sti lukkes, fjernes dens stielement fra stilisten.

Den del af træet, der arbejdes på, repræsenteres af variablen DELTRAE. Hvis der kun arbejdes på en enkelt sti, indeholder DELTRAE en pointer til det pågældende stielement, og hvis der arbejdes på hele træet, har DELTRAE værdien NIL.

Repræsentation af strategi

De strategier, der defineres v. hj. a. ordren DEF, repræsenteres v. hj. a. et array for hver af strategiens parametre:

```
FORM  : array [1..5] of formparam ;
VIDERE: array [1..5] of vidparam ;
NYTUR:  array [1..5] of nyparam ;
STOP  : array [1..5] of integer ;
```

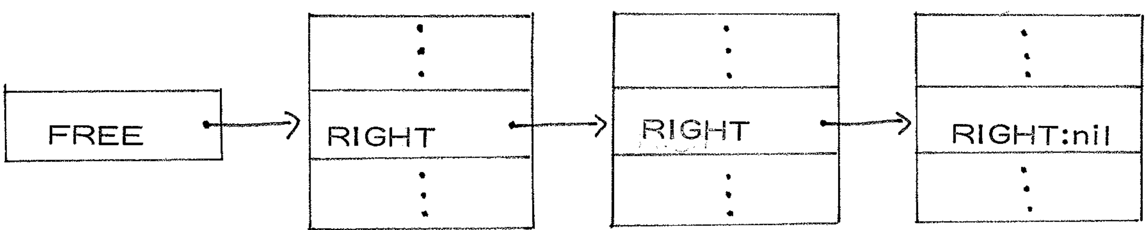
hvor skalartyperne er defineret ved:

```
formparam = (numre, prnavn, udengren, alle, tom) ;
vidparam  = (nej, tilgren, allenye) ;
nyparam   = (foerst, sidst) ;
```

FORM [n] = tom repræsenterer, at strategi nr n ikke er defineret.

Lagerallokering

Da datarepræsentationen er valgt således, at der kun bruges én recordtype, bliver lagerallokeringen meget enkel. Ved starten af programmet anbringes alle de records, der er til rådighed (dvs. alle komponenterne i klassen lager: class max of elem)på en fri liste, som refereres til af variablen FREE:



Når der skal bruges en record, hentes denne fra den frie liste v. hj. a. proceduren HENTELEM, og når en record ikke længere spiller nogen rolle i datastrukturen, leveres den straks tilbage til den frie liste v. hj. a. proceduren GEMELEM.

Da Pascal ikke har dynamisk lagerallokering, er antal komponenter i klassen lager fastlagt ved konstanten MAX. For at brugeren ikke skal til at ændre i selve programmet, hvis han ønsker mere eller mindre lagerkapacitet, er der som permanent files i systemet lagret flere forskellige versioner af programmet, der kun adskiller sig fra hinanden ved værdien af MAX (500, 1000, 2000, 3000).

Udvikling af en sætning

Hovedelementet i en beregning er udviklingen af en sætning i træet og den dertil hørende udvidelse af træet. Hvis sætningen skal udvikles v. hj. a. en af inferensreglerne i figur 4, sker dette v. hj. a. proceduren UDVIKL, hvis virkemåde kan skitseres i flg. trin:

1: Først opbygges recordstrukturer svarende til de nye sætninger, som kan dannes ved anvendelse af den pågældende regel. Hvis sætningen, der skal udvikles, f. eks. er af formen $A \rightarrow B$, dannes der en repræsentation af sætningen $\neg A$ og en repræsentation af sætningen B. Hvis A selv er af formen $\neg A^*$, anvendes straks reglen for dobbelt negation, og $\neg A$ repræsenteres altså som A^*.

2: Hvis der arbejdes på en enkelt sti, tilføjes de nye sætninger til stiens blad v. hj. a. proceduren TILFØJ1, TILFØJ2 eller TILFØJ3, afhængig af om den nye struktur er af

type 1: ↓□

type 2: ↓□↓□ eller

type 3: ↙↘ □ □

Hvis der arbejdes på hele træet, kaldes proceduren NED, som finder alle blade under den sætning, der er under udvikling, og til alle disse blade føjer en kopi af den nye struktur v. hj. a. TILFØJ1, TILFØJ2 eller TILFØJ3. Hvis samme sætning indgår flere steder i træet, er hver forekomst altså repræsenteret ved sin egen recordstruktur, således at hvis den ene forekomst senere slettes, kan den tilhørende struktur tilbagegives til den frie liste, uden at de andre forekomster berøres.

3: Hvis der arbejdes på hele træet, slettes den sætning, der er blevet udviklet.

Hvis en sætning skal udvikles v. hj. a. reglen for alkvantor, sker dette på helt tilsvarende måde v. hj. a. proceduren UDVIKLAL, idet pkt. 3 dog udelades, og det samme er tilfældet, hvis reglen for lighedstegn skal anvendes som følge af ordren SUBV eller SUBH, idet substitutionen foretages v. hj. a. proceduren SKIFT.

Før TILFØJ1, TILFØJ2 og TILFØJ 3 tilføjer en ny struktur til en sti, undersøger de v. hj. a. funktionen TEST, om de nye sætninger er overflødige, eller om de evt. bevirker, at den pågældende sti kan lukkes, Funktionen TEST har som argumenter en pointer til stiens blad og en pointer til en ny sætning, og TEST får værdien "overflødig", hvis den nye sætning er af formen t = t, eller hvis den allerede findes i stien, værdien "modstrid", hvis den nye sætning er af formen t ≠ t, eller hvis dens negation findes i stien, og værdien "nytblad" ellers. Hvis den nye struktur f. eks. er af type 3, kan anvendelsen af TEST illustreres således:

Først findes værdien af TEST for begge de sætninger, der skal tilføjes som en forgrening, og flg. tilfælde kan da indtræffe:

1: Begge værdier = "modstrid" : proceduren SLETGREN kaldes.

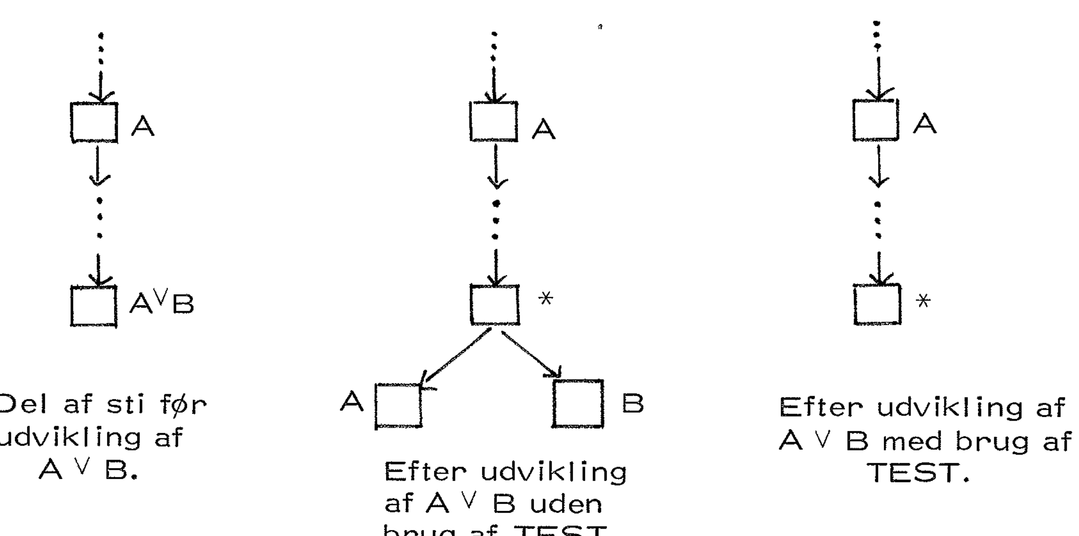
2: Begge værdier = "nytblad" : en forgrening tilføjes bladet.

3: Begge værdier = "overflødig" : ingen tilføjelser foretages.

4: En værdi = "nytblad" og én værdi : "modstrid" : kun én sætning tilføjes stien (ingen nye stier).

5: En værdi = "overflødig" og én værdi : "modstrid" : ingen tilføjelser foretages.

6: En værdi = "nytblad" og én værdi = "overflødig" : ingen tilføjelser foretages, da en lukning af den oprindelige sti er ensbetydende med (og nemmere end) en lukning af begge de stier, som fremkommer ved forgreningen. (se flg. figur)



Del af sti før udvikling af A V B.

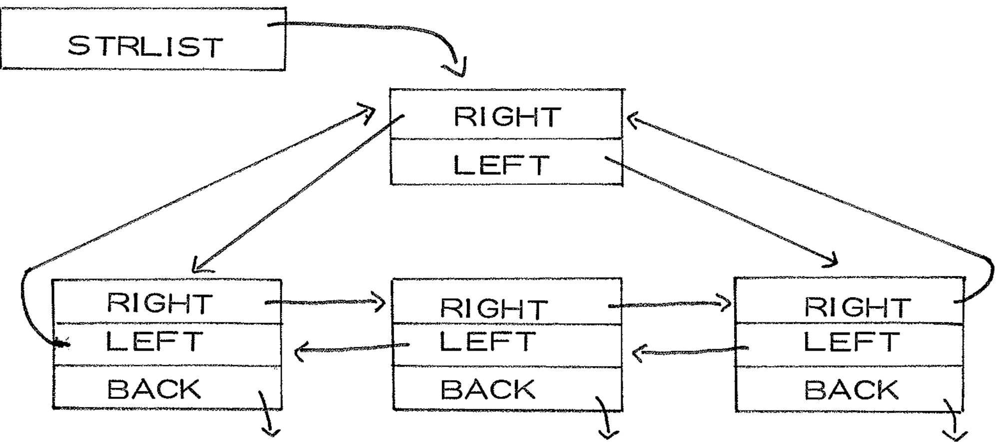
Efter udvikling af A V B uden brug af TEST.

Efter udvikling af A V B med brug af TEST.

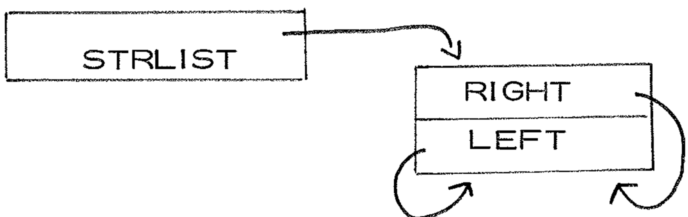
Ved på denne måde at foretage en evt. reduktion af træet samtidig med tilføjelsen af nye sætninger opnås, at træet til ethvert tidspunkt er minimalt i den forstand, at det indeholder færrest mulige sætninger og stier.

Udførelse af strategier

Til at udpege de sætninger, der skal udvikles under udførelsen af en strategi, anvendes en cirkulær liste af records. I disse records indeholder feltet back en pointer til en knude i træet, der repræsenterer en sætning, som skal udvikles iflg. strategien. Hovedet for denne liste er en speciel record, som refereres til af den globale variabel STRLIST:



Når programmet ikke er ved at udføre en strategi er listen tom:



og ved kommandoen GO, der kalder proceduren GOSTR, anvendes listen på flg. måde:

Først anbringes på listen referencer til alle de sætninger, i træet, der iflg. strategiens parameter "formler" skal udvikles. Hvis værdien af "formler" er "numre", og brugeren derfor bedes om at ind-

taste en række sætningsnumre, findes de pågældende sætninger i træet v. hj. a. proceduren FINDFORM. For de tre andre værdier, som "formler" kan have, gælder det, at sætningerne i træet udvælges v. hj. a. proceduren FORMSI, som løber ned gennem hele træet og for hver knude v. hj. a. funktionen FORMMED afgør, om den pågældende sætning skal tilføjes strategilisten eller ej.

Når strategilisten således er initialiseret, udføres strategien ved, at sætningerne udvikles i den rækkefølge, de optræder på listen. Når en sætning er udviklet, fjernes den tilsvarende record fra strategilisten, og hvis strategiens parameter "videre" er forskellig fra "nej", hvilket markeres ved, at den globale boolske variabel CHECKNY får værdien true ved starten af GOSTR, så tilføjes strategilisten evt. records, som refererer til de nye sætninger, der er blevet tilføjet træet ved udviklingen af den pågældende sætning. Dette sker ved, at procedurerne TILFØJ1, TILFØJ2 og TILFØJ3, såfremt CHECKNY = true, kalder proceduren NYMED, hver gang en sætning er blevet tilføjet træet. NYMED undersøger, om den nye sætning skal tilføjes strategilisten iflg. strategiens parameter "videre", og hvis dette er tilfældet, sker tilføjelsen først eller sidst på listen afhængig af parameteren "nytur".

Dersom en sti lukkes under udførelsen af en strategi, kan det ske, at en af de knuder, der refereres til af en record i strategilisten, fjernes fra træet, Om dette er tilfældet, undersøges af proceduren SLETGREN, der i givet fald fjerner den pågældende record fra listen. Det er bl. a. i denne situation, at det er nyttigt, at strategilisten er organiseret som en cirkulær liste.

Denne proces fortsætter nu, indtil strategilisten er tom, eller indtil brugerens afslutter udførelsen af strategien ved en af de afbrydelser, der sker, såfremt parameteren "stopinterval" er forskellig fra nul.

Input/Output

Indlæsning og udskrivning af sætninger sker v. hj. a. procedurerne READFORMEL og WRITEFORMEL. READFORMEL, der ind-

læser de forskellige dele af en sætning v.hj.a. hjælpeprocedurerne READTERM, READKVANTOR, READLIGHED og READPRFKT (prædikater og funktioner), analyserer den givne sætning og giver fejludskrift, hvis sætningsstrukturen ikke er i overensstemmelse med den definerede syntax. Samtidig opbygges intern repræsentation af sætningen i overensstemmelse med den tidligere beskrevne datarepræsentation. Ved denne konvertering fra inputformat til intern datastruktur forsvinder eventuelle overflødige paranteser, ligesom dobbeltnegationer elimineres, så at sætningen, når den udskrives v.hj.a. WRITEFORMEL, får så simpelt et udseende som muligt. Analysen af sætningen er dog ikke fuldstændig, idet det ikke undersøges, om der forekommer frie variable, eller om forskellige kvantorer har samme variabel i modstrid med vores konvention, ligesom det heller ikke undersøges, om samme prædikat- eller funktionssymbol optræder med forskellige antal variable.

Til kontrol af hvordan programmet har fortolket og repræsenteret inputsætningen, udskrives sætningen straks på skærmen v.hj.a. WRITEFORMEL. Den forenkling af sætningens udseende, der herved kan ske, illustreres ved et par eksempler:

1: Input : – (– Ax1 (P[x1])).
Output : Ax1 P[x1].

2: Input : – (c1 = c2) → (Ex1 (Ax2 (Q ∨ (x1 = x2)))).
Output : c1 ≠ c2 → Ex1 Ax2 (Q ∨ x1 = x2).

Selve programkroppen består i det væsentlige af en case-konstruktion, i hvilken der til hver interaktiv kommando svarer en (sammensat) sætning, som aktiverer de procedurer, der sørger for udførelsen af ordren. Indlæsningen af kommandoordet sker ved at læse ordet karakter for karakter, indtil der er indlæst tilstrækkelig mange bogstaver til at skelne ordet fra de øvrige kommandoer. Derefter skippes resten af ordet. Det betyder, at man ikke behøver at skrive kommandoerne helt ud, men f.eks. kan nøjes med at skrive F eller FOC (men altså også FIP) i stedet for FOCUS.

Det bemærkes, at der i programmet optræder procedurer, der kun kaldes et enkelt sted i programmet, f. eks. INITIALISERING og GOSTR. Dette skyldes, at såfremt procedurekroppene i stedet indarbejdes i programkroppen, erklærer pascalcompileren, at programkroppen er for lang.

AFSNIT 5

EKSEMPLER

På de følgende sider er vist nogle eksempler på kørsler med programmet. I disse eksempler er dialogen mellem bruger og program, der jo normalt foregår via en dataskærm, gengivet v.hj.a. en speciel batch-version af programmet. De beviste teoremer er alle meget enkle og skal blot tjene til illustration af programmets virkemåde, mens der i næste afsnit forsøges en vurdering af metodens og programmets muligheder og begrænsninger.

Eksempel 1

Her bevises teoremet:

$$\forall x1\ (P[x1,\ c2] \rightarrow P[c1,\ c2],$$
$$\neg P\ [c1,\ c2]$$
$$\vdash\quad \neg\exists\ x1\ \ P\ [x1,\ c2]$$

Dette eksempel er det samme, som på figur 5 er gennemregnet ved håndkraft, og ved at sammenholde programudskriften med figur 5 ses tydeligt den "minimalisering" af træet, som er indbygget i programmet, og som blev diskuteret i foregående afsnit. De sætninger, som på figur 5 har numrene 8, 9, 11, 12 og 13, bliver således aldrig tilføjet træet, da de enten er overflødige eller forårsager lukning af stier.

Eksempel 2

Her bevises gyldigheden af sætningen:

$$\forall x1\ \ \exists x2\ \ \forall x3\ \ ((F[x1] = x3 \rightarrow x3 = x2) \wedge (x3 = x2 \rightarrow F[x1] = x3)).$$

Dette eksempel er det samme, som er vist på figur 6, og de to versioner ses at stemme nøje overens.

Eksempel 3

I dette eksempel bevises et teorem fra den rene sætningskalkyle:

$$((P \rightarrow Q) \rightarrow P) \rightarrow P$$

$$\vdash (((P \vee Q) \wedge (P \rightarrow R)) \wedge (Q \rightarrow R)) \rightarrow R.$$

Eksempel 4

Dette eksempel beviser, at flg. sætning er gyldig:

$$\exists x1 \ \forall x2 \ P[x1, x2] \rightarrow \forall x2 \ \exists x1 \ P[x1, x2].$$

Eksempel 5

I dette eksempel bevises et teorem fra gruppeteorien, nemlig forkortningsreglen. Hvis der anvendes notationen F[x, y] for kompositionen af elementerne x og y, I[x] for det inverse element til x og betegnelsen c1 for gruppens nulelement, så kan teoriens axiomer fo formuleres:

$$\forall x1 \ \forall x2 \ \forall x3 \ F[x1, F[x2, x3]] = F[F[x1, x2], x3]$$

$$\forall x1 \ F[x1, c1] = x1$$

$$\forall x1 \ F[x1, I[x1]] = c1.$$

I eksemplet bevises, at der ud fra disse axiomer logisk følger sætningen:

$$\forall x1 \ \forall x2 \ \forall x3 \ (F[x1, x3] = F[x2, x3] \rightarrow x1 = x2).$$

Det ses, at der på grund af de mange forekomster af lighedstegnet stilles større krav til brugerens aktive medvirken end i de foregående eksempler.

EKSEMPEL 1.

```
TAST ANTAL STARTFORMLER
↱↱ 3
TAST FORMLER, AFSLUT HVER FORMEL MED $
↱↱ AX1 (P[X1,C2]↱P[C1,C2]) $
   -P[C1,C2] $
   EX1 P[X1,C2] $

STI 1, FORMEL 1: AX1 (P[X1,C2]↱P[C1,C2])
STI 1, FORMEL 2: -P[C1,C2]
STI 1, FORMEL 3: EX1 P[X1,C2]

TAST NY ORDRE
↱↱ FOCUS 3
STI 1, FORMEL 4: P[C3,C2]

TAST NY ORDRE
↱↱ ALKV C1 C2 C3
TAST FORMELNR
↱↱ 1
STI 1, FORMEL 5: P[C1,C2]↱P[C1,C2]
STI 1, FORMEL 6: P[C2,C2]↱P[C1,C2]
STI 1, FORMEL 7: P[C3,C2]↱P[C1,C2]

TAST NY ORDRE
↱↱ FOCUS 5 6 7
STI 1, FORMEL 8: -P[C2,C2]
STI NR 1 LUKKES

TEOREMET ER BEVIST


TAST NY ORDRE
↱↱ NY
MAX ANTAL BRUGTE RECORDS: 31
```

```
TAST ANTAL STARTFORMLER
→→ 1
TAST FORMLER, AFSLUT HVER FORMEL MED $
→→ -AX1 EX2 AX3 ((F[X1]=X3→X3=X2)∧(X3=X2→F[X1]=X3)) $

STI 1, FORMEL 1: -AX1 EX2 AX3 ((F[X1]=X3→X3=X2)∧(X3=X2→F[X1]=X3))

TAST NY ORDRE
→→ DEF 1
FORMLER
→→ ALLE
VIDERE
→→ ALLENYE
NYTUR
→→ SIDST
STOPINTERVAL
→→ 0

STRATEGI NR: 1
FORMLER: ALLE
VIDERE: ALLENYE
NYTUR: SIDST
STOPINTERVAL: 0

TAST NY ORDRE
→→ GO 1
STI 1, FORMEL 2: EX1 -EX2 AX3 ((F[X1]=X3→X3=X2)∧(X3=X2→F[X1]=X3))
STI 1, FORMEL 3: -EX2 AX3 ((F[C1]=X3→X3=X2)∧(X3=X2→F[C1]=X3))
STI 1, FORMEL 4: AX2 -AX3 ((F[C1]=X3→X3=X2)∧(X3=X2→F[C1]=X3))

TAST NY ORDRE
→→ STI 1
STI NR 1 BESTAAR AF:
FORMEL 4: AX2 -AX3 ((F[C1]=X3→X3=X2)∧(X3=X2→F[C1]=X3))

TAST NY ORDRE
→→ ALKV F[C1]
TAST FORMELNR
→→ 4
STI 1, FORMEL 5: -AX3 ((F[C1]=X3→X3=F[C1])∧(X3=F[C1]→F[C1]=X3))

TAST NY ORDRE
→→ GO 1
STI 1, FORMEL 6: EX3 -((F[C1]=X3→X3=F[C1])∧(X3=F[C1]→F[C1]=X3))
STI 1, FORMEL 7: -((F[C1]=C2→C2=F[C1])∧(C2=F[C1]→F[C1]=C2))
STI 1, FORMEL 8: -(F[C1]=C2→C2=F[C1])
STI 2, FORMEL 9: -(C2=F[C1]→F[C1]=C2)
STI 1, FORMEL 10: F[C1]=C2
STI NR 1 LUKKES

STI 2, FORMEL 11: C2=F[C1]
STI NR 2 LUKKES

TEOREMET ER BEVIST


TAST NY ORDRE
→→ BYE
BYE,BYE
MAX ANTAL BRUGTE RECORDS: 58
```

EKSEMPEL 3.

```
TAST ANTAL STARTFORMLER
→→ 2
TAST FORMLER, AFSLUT HVER FORMEL MED $
→→ ((P→Q)→P)→P $
   -((((PvQ)∧(P→R))∧(Q→R))→R) $

STI 1, FORMEL 1: ((P→Q)→P)→P
STI 1, FORMEL 2: -((((PvQ)∧(P→R))∧(Q→R))→R)

TAST NY ORDRE
→→ DEF 1
FORMLER
→→ ALLE
VIDERE
→→ ALLENYE
NYTUR
→→ SIDST
STOPINTERVAL
→→ 0

STRATEGI NR: 1
FORMLER: ALLE
VIDERE: ALLENYE
NYTUR: SIDST
STOPINTERVAL: 0

TAST NY ORDRE
→→ GO 1
STI 1, FORMEL 3: -((P→Q)→P)
STI 2, FORMEL 4: P
STI 1, FORMEL 5: ((PvQ)∧(P→R))∧(Q→R)
STI 1, FORMEL 6: -R
STI 2, FORMEL 7: ((PvQ)∧(P→R))∧(Q→R)
STI 2, FORMEL 8: -R
STI 1, FORMEL 9: P→Q
STI 1, FORMEL 10: -P
STI 1, FORMEL 11: (PvQ)∧(P→R)
STI 1, FORMEL 12: Q→R
STI 2, FORMEL 13: (PvQ)∧(P→R)
STI 2, FORMEL 14: Q→R
STI 1, FORMEL 15: PvQ
STI 1, FORMEL 16: P→R
STI 1, FORMEL 17: -Q
STI 2, FORMEL 18: PvQ
STI 2, FORMEL 19: P→R
STI 2, FORMEL 20: -Q
STI NR 1 LUKKES

STI NR 2 LUKKES

TEOREMET ER BEVIST


TAST NY ORDRE
→→ NY
MAX ANTAL BRUGTE RECORDS: 74
```

EKSEMPEL 4.

```
TAST ANTAL STARTFORMLER
↱↱ 1
TAST FORMLER, AFSLUT HVER FORMEL MED $
↱↱ -(EX1 AX2 P[X1,X2]↱AX2 EX1 P[X1,X2]) $

STI 1, FORMEL 1: -(EX1 AX2 P[X1,X2]↱AX2 EX1 P[X1,X2])

TAST NY ORDRE
↱↱ FOCUS 1
STI 1, FORMEL 2: EX1 AX2 P[X1,X2]
STI 1, FORMEL 3: -AX2 EX1 P[X1,X2]

TAST NY ORDRE
↱↱ FOCUS 2 3
STI 1, FORMEL 4: AX2 P[C1,X2]
STI 1, FORMEL 5: EX2 -EX1 P[X1,X2]

TAST NY ORDRE
↱↱ FOCUS 5
STI 1, FORMEL 6: -EX1 P[X1,C2]

TAST NY ORDRE
↱↱ FOCUS 6
STI 1, FORMEL 7: AX1 -P[X1,C2]

TAST NY ORDRE
↱↱ STI 1
STI NR 1 BESTAAR AF:
FORMEL 7: AX1 -P[X1,C2]
FORMEL 4: AX2 P[C1,X2]

TAST NY ORDRE
↱↱ ALKV C1 C2
TAST FORMELNR
↱↱ 7 4
STI 1, FORMEL 8: -P[C1,C2]
STI 1, FORMEL 9: -P[C2,C2]
STI 1, FORMEL 10: P[C1,C1]
STI NR 1 LUKKES

TEOREMET ER BEVIST


TAST NY ORDRE
↱↱ NY
MAX ANTAL BRUGTE RECORDS: 22
```

EKSEMPEL 5.

```
TAST ANTAL STARTFORMLER
→→ 4
TAST FORMLER, AFSLUT HVER FORMEL MED $
→→ AX1 AX2 AX3 F[X1,F[X2,X3]]=F[F[X1,X2],X3] $
   AX1 F[X1,C1]=X1 $
   AX1 F[X1,I[X1]]=C1 $
   -AX1 AX2 AX3 (F[X1,X3]=F[X2,X3]→X1=X2) $

STI 1, FORMEL 1: AX1 AX2 AX3 F[X1,F[X2,X3]]=F[F[X1,X2],X3]
STI 1, FORMEL 2: AX1 F[X1,C1]=X1
STI 1, FORMEL 3: AX1 F[X1,I[X1]]=C1
STI 1, FORMEL 4: -AX1 AX2 AX3 (F[X1,X3]=F[X2,X3]→X1=X2)

TAST NY ORDRE
→→ FOCUS 4
STI 1, FORMEL 5: EX1 -AX2 AX3 (F[X1,X3]=F[X2,X3]→X1=X2)

TAST NY ORDRE
→→ FOCUS 5
STI 1, FORMEL 6: -AX2 AX3 (F[C2,X3]=F[X2,X3]→C2=X2)

TAST NY ORDRE
→→ FOCUS 6
STI 1, FORMEL 7: EX2 -AX3 (F[C2,X3]=F[X2,X3]→C2=X2)

TAST NY ORDRE
→→ FOCUS 7
STI 1, FORMEL 8: -AX3 (F[C2,X3]=F[C3,X3]→C2=C3)

TAST NY ORDRE
→→ FOCUS 8
STI 1, FORMEL 9: EX3 -(F[C2,X3]=F[C3,X3]→C2=C3)

TAST NY ORDRE
→→ FOCUS 9
STI 1, FORMEL 10: -(F[C2,C4]=F[C3,C4]→C2=C3)

TAST NY ORDRE
→→ FOCUS 10
STI 1, FORMEL 11: F[C2,C4]=F[C3,C4]
STI 1, FORMEL 12: C2≠C3

TAST NY ORDRE
→→ STI 1
STI NR 1 BESTAAR AF:
FORMEL 12: C2≠C3
FORMEL 11: F[C2,C4]=F[C3,C4]
FORMEL 3: AX1 F[X1,I[X1]]=C1
FORMEL 2: AX1 F[X1,C1]=X1
FORMEL 1: AX1 AX2 AX3 F[X1,F[X2,X3]]=F[F[X1,X2],X3]

TAST NY ORDRE
→→ ALKV C2 C3
TAST FORMELNR
→→ 1
STI 1, FORMEL 13: AX2 AX3 F[C2,F[X2,X3]]=F[F[C2,X2],X3]
STI 1, FORMEL 14: AX2 AX3 F[C3,F[X2,X3]]=F[F[C3,X2],X3]
```

```
TAST NY ORDRE
↱↱ ALKV C4
TAST FORMELNR
↱↱ 13 14
STI 1, FORMEL 15: AX3 F[C2,F[C4,X3]]=F[F[C2,C4],X3]
STI 1, FORMEL 16: AX3 F[C3,F[C4,X3]]=F[F[C3,C4],X3]

TAST NY ORDRE
↱↱ ALKV I[C4]
TAST FORMELNR
↱↱ 15 16
STI 1, FORMEL 17: F[C2,F[C4,I[C4]]]=F[F[C2,C4],I[C4]]
STI 1, FORMEL 18: F[C3,F[C4,I[C4]]]=F[F[C3,C4],I[C4]]

TAST NY ORDRE
↱↱ STI 1
STI NR 1 BESTAAR AF:
FORMEL 18: F[C3,F[C4,I[C4]]]=F[F[C3,C4],I[C4]]
FORMEL 17: F[C2,F[C4,I[C4]]]=F[F[C2,C4],I[C4]]
FORMEL 16: AX3 F[C3,F[C4,X3]]=F[F[C3,C4],X3]
FORMEL 15: AX3 F[C2,F[C4,X3]]=F[F[C2,C4],X3]
FORMEL 14: AX2 AX3 F[C3,F[X2,X3]]=F[F[C3,X2],X3]
FORMEL 13: AX2 AX3 F[C2,F[X2,X3]]=F[F[C2,X2],X3]
FORMEL 12: C2≠C3
FORMEL 11: F[C2,C4]=F[C3,C4]
FORMEL 3: AX1 F[X1,I[X1]]=C1
FORMEL 2: AX1 F[X1,C1]=X1
FORMEL 1: AX1 AX2 AX3 F[X1,F[X2,X3]]=F[F[X1,X2],X3]

TAST NY ORDRE
↱↱ SUBV 11
TAST FORMELNR
↱↱ 17
STI 1, FORMEL 19: F[C2,F[C4,I[C4]]]=F[F[C3,C4],I[C4]]

TAST NY ORDRE
↱↱ ALKV C4
TAST FORMELNR
↱↱ 3
STI 1, FORMEL 20: F[C4,I[C4]]=C1

TAST NY ORDRE
↱↱ SUBV 20
TAST FORMELNR
↱↱ 19 18 17
STI 1, FORMEL 21: F[C2,C1]=F[F[C3,C4],I[C4]]
STI 1, FORMEL 22: F[C3,C1]=F[F[C3,C4],I[C4]]
STI 1, FORMEL 23: F[C2,C1]=F[F[C2,C4],I[C4]]

TAST NY ORDRE
↱↱ SUBH 21
TAST FORMELNR
↱↱ 22
STI 1, FORMEL 24: F[C3,C1]=F[C2,C1]
```

```
TAST NY ORDRE
↱↱ STI 1
STI NR 1 BESTAAR AF:
FORMEL 24: F[C3,C1]=F[C2,C1]
FORMEL 23: F[C2,C1]=F[F[C2,C4],I[C4]]
FORMEL 22: F[C3,C1]=F[F[C3,C4],I[C4]]
FORMEL 21: F[C2,C1]=F[F[C3,C4],I[C4]]
FORMEL 20: F[C4,I[C4]]=C1
FORMEL 19: F[C2,F[C4,I[C4]]]=F[F[C3,C4],I[C4]]
FORMEL 18: F[C3,F[C4,I[C4]]]=F[F[C3,C4],I[C4]]
FORMEL 17: F[C2,F[C4,I[C4]]]=F[F[C2,C4],I[C4]]
FORMEL 16: AX3 F[C3,F[C4,X3]]=F[F[C3,C4],X3]
FORMEL 15: AX3 F[C2,F[C4,X3]]=F[F[C2,C4],X3]
FORMEL 14: AX2 AX3 F[C3,F[X2,X3]]=F[F[C3,X2],X3]
FORMEL 13: AX2 AX3 F[C2,F[X2,X3]]=F[F[C2,X2],X3]
FORMEL 12: C2≠C3
FORMEL 11: F[C2,C4]=F[C3,C4]
FORMEL 3: AX1 F[X1,I[X1]]=C1
FORMEL 2: AX1 F[X1,C1]=X1
FORMEL 1: AX1 AX2 AX3 F[X1,F[X2,X3]]=F[F[X1,X2],X3]

TAST NY ORDRE
↱↱ ALKV C2 C3
TAST FORMELNR
↱↱ 2
STI 1, FORMEL 25: F[C2,C1]=C2
STI 1, FORMEL 26: F[C3,C1]=C3

TAST NY ORDRE
↱↱ SUBV 26
TAST FORMELNR
↱↱ 24
STI 1, FORMEL 27: C3=F[C2,C1]

TAST NY ORDRE
↱↱ SUBV 25
TAST FORMELNR
↱↱ 27
STI NR 1 LUKKES

TEOREMET ER BEVIST


TAST NY ORDRE
↱↱ BYE
BYE,BYE
MAX ANTAL BRUGTE RECORDS: 179
```

AFSNIT 6

VURDERING OG KOMMENTAR

I dette sidste afsnit skal videregives nogle af de erfaringer, som jeg har gjort under arbejdet med den her beskrevne theoremprover, ligesom jeg vil forsøge at give en vurdering af det foreliggende programs muligheder og begrænsninger.

Med hensyn til selve programmeringsarbejdet kan det roligt siges, at det er forløbet alt andet end gnidningsløst. Da jeg oprindelig planlagde opgaven, var det meningen, at programmet skulle skrives i Algol, som det mest velegnede af de sprog, der på det tidspunkt var tilgængelige på RECAU's CDC 6400. Da den første version af programmet (ca. 1400 linier), som var en foreløbig udgave uden en række af de faciliteter, der er med i det foreliggende program, var klar, blev arbejdet forsinket af fejl i algolcompileren, der bl.a. forhindrede interaktivt input/output. Efter at disse fejl var udbedret, og indkøringen af programmet kunne begynde, viste det sig, at algolcompilerens effektivitet var så ringe, at det oversatte program ikke kunne køre på 50000_8 cm-ord, hvorfor det var håbløst at tænke på interaktiv kørsel, som jo er en afgørende forudsætning for det planlagte system. Hele programmet måtte derfor kasseres, og da en pascalcompiler i mellemtiden var blevet indkørt på maskinen, og de første erfaringer med denne var særdeles gode, valgte jeg at skrive programmet om i Pascal. Resultatet var, at den første pascalversion, der i formåen svarede ret nøje til det kasserede algolprogram, kunne køre på under 16000_8 cm-ord. Forklaringen på denne enorme forbedring skal først og fremmest søges i den langt mere effektive compiler, men også i det forhold, at jeg ved omprogrammeringen naturligvis udnyttede erfaringerne fra det første arbejde til en ændring og forenkling af dele af programmets struktur. Hertil kommer, atpascals datastrukturfaciliteter muliggør en langt mere bekvem datarepræsentation, og Pascals overlegenhed til opgaver af netop denne type blev til fulde demonstreret af den enkle måde, hvorpå det i dette sprog er muligt at programmere ting, som kun med besvær lader sig formulere i CDC-algol.

Takket være udnyttelsen af datatypen "packed record" er den lagerplads, som kræves til repræsentation af sætningstræet, langt mindre end i CDC-algol, som ikke har indbyggede bitmanipulationsmekanismer. Af de foregående eksempler ses, at det antal records, der behøves til at repræsentere et sætningstræ, er særdeles beskedent, og da hver af disse records kun fylder to ord, bliver kravene til lagerkapacitet for selv ret store problemer meget overkommelige. Med et maximalt antal reocrds på 500 kan det nuværende program køre på 20000_8 cm-ord, og med 3000 records til rådighed er pladskravet 32000_8 cm-ord. Denne effektive pladsudnyttelse sker vel at mærke ikke på bekostning af regnehastigheden. Wirth anfører selv, at anvendelsen af packed record kun medfører et merforbrug af cp-tid på 2%, og det kan oplyses, at alle eksemplerne i foregående afsnit er kørt med et tidsforbrug på mindre end 1 cp-sekund. Ved interaktiv kørsel fra en terminal betyder det ringe pladskrav derimod, at responstiden bliver lille, så dialogen mellem program og bruger forløber uden generende pauser.

Af programmeringstekniske ting skal endvidere gøres opmærksom på, at de enkelte dele af programmets virkemåde er holdt mest muligt adskilt fra hinanden, selv om dette muligvis koster noget i effektivitet. Dette skyldes, at det foreliggende program ikke foregiver at være en "færdig" theorem-prover, men at det tværtimod nemt skal kunne ændres og udvides, efterhånden som erfaringerne med det gør det ønskeligt. I en mere endelig version ville man formentlig også ønske at gøre lidt mere ud af udenomsværkerne: mere udførlige fejludksrifter, mere beskrivende funktionsnavne osv.

Der er flere grunde til, at jeg har valgt at basere min theorem-prover på træmetoden fremfor resolutionsmetoden. For det første er der i de sidste 6-7 år blevet investeret så megen man-power i udforskningen af resolutionsprincippets anvendelsesmuligheder, og man har opnået så raffinerede teoretiske og praktiske resultater, at det i et arbejde af dette omfang ikke er sandsynligt, at man kan få fat i en ende af problemet, som ikke er grundigt undersøgt og gennemprøvet tidligere. Jeg har derfor følt det mere interessant at starte på bar bund med en ny metode, vel vidende at man på denne måde naturligvis ikke kan gøre sig håb om at nå et praktisk resultat, der i effektivitet kan måle sig med f. eks. den i afsnit 2

beskrevne theorem-prover. Alligevel synes jeg, at træmetoden rummer nogle muligheder, som fortjener at blive gennemprøvet ved experimenter. Det centrale for mig har været at lave et system, der i kraft af sine interaktive faciliteter giver brugeren mulighed for selv at experimentere med forskellige metoder til løsning af et forelagt problem, og hvis man skal have nogen glæde af en sådan interaktion mellem menneske og maskine, er det væsentligt, at kommunikationen sker i en notation, der afviger så lidt som muligt fra den "naturlige" notation inden for det pågældende problemområde, i dette tilfælde prædikatkalkylen. I denne henseende mener jeg, at træmetoden har et afgjort fortrin frem for resolutionsmetoden, idet den ikke kræver, at sætningerne først omskrives til klausulform, hvorved man meget nemt mister den intuitive fornemmelse af sætningernes "indhold", så beregningerne bliver en abstrakt symbolmanipulation. Når dertil kommer, at træmetodens regler er så overordentlig simple, tror jeg, at det implementerede system vil være anvendeligt for folk uden datalogisk baggrund, men blot med kendskab til elementær logik.

Ved anvendelse af programmet er det afgørende problem jo at begrænse væksten af træet ved at udvikle de sætninger, som hurtigst fører til lukning af alle stier. For sætningskalkylen er problemet ikke stort, idet de nye sætninger bliver simplere og simplere, samtidig med at de udviklede sætninger kan slettes. Man vil derfor så godt som altid som i eks. 3 kunne klare sig med den mindst restriktive strategi og alligevel hurtigt nå et resultat.

Vanskeligere er det, hvis sætningerne indeholder mange kvantorer, idet den fuldstændige metode, hvor alle alkvantorer udvikles v.hj.a. alle konstante termer, meget hurtigt kan blive uoverkommelig. Jeg har ikke forsøgt at indbygge en strategi i programmet, der afgør hvilke konstante termer, der skal anvendes ved udviklingen af de sætninger, som styres af en alkvantor, idet jeg tror, at det er på dette punkt, det interaktive element har sin afgørende værdi. Den menneskelige intuition vil formentlig være de fleste mekaniske procedurer overlegen, når det drejer sig om ud fra udseendet af træets øvrige sætninger at afgøre hvilke konstante termer, der er mest lovende i en given situation. Det er muligt, at man ved længere tids experimenteren med programmet kan få en række er-

faringer om, efter hvilke kriterier det er mest nyttigt at udvikle alkvantorerne og på den måde få inspiration til en automatisering af denne proces. Dette vil selvfølgelig være en meget ønskelig udvidelse af programmet, men jeg tror, det er mest frugtbart at basere sådanne strategier på et større erfaringsmateriale fremfor på forhåndsteoretiske overvejelser. I sin nuværende form er programmet velegnet til sådanne praktiske experimenter. Man kan f. eks. starte med at udvikle de forekommende alkvantorer v.hj.a. nogle enkelte konstanter og så lade programmet automatisk udføre en række mere trivielle skridt ved en mere eller mindre restriktiv strategi, hvor det også kan være nyttigt at skifte mellem at arbejde på en enkelt sti og på hele træet. Dernæst kastes nogle nye sætninger ind i spillet ved at udvikle alkvantorer v.hj.a. nye konstante termer. De nye sætninger kan så videreudvikles automatisk, nye termer bringes i anvendelse osv. Ved denne fremgangsmåde er det med lethed lykkedes at bevise en række teoremer hentet fra elementære lærebøger i logik.

Derimod har de første erfaringer med programmet vist, at det i sin nuværende form er utilstrækkeligt til en effektiv behandling af lighedstegn. Hvis der i sætningstræet optræder flere lighedstegn, bliver det mulige antal substitutioner hurtigt så stort, at det bliver vanskeligt for brugeren at bevare overblikket og dirigere udviklingen af træet. Teoremerne skal ikke være meget mere komplicerede end det i eks. 5 beviste, før processen føles meget tung. Hvis programmet med udbytte skal kunne finde anvendelse på ikke helt simple teoremer fra f. eks. gruppeteori eller talteori, er der derfor behov for indførelse af faciliteter, der fritager brugeren for at skulle specificere hver enkelt substitution. Som eksempel på sådanne ønskelige forbedringer kan det nævnes, at det ville være rimeligt at forvente, at programmet i eks. 5 selv kunne finde en modstrid ved at sammenholde sætningerne 24, 25 og 26 med sætning nr. 12. Det gælder også her, at forbedringer af programmets formåen ved udvidelser af denne art bedst opnås gennem praktiske erfaringer med programmet, og på dette punkt er jeg ikke nået så langt, som jeg havde ønsket, først og fremmest på grund af de tidrøvende komplikationer under programmeringen.

Sammenfattende mener jeg, at det kan siges, at det foreliggende program har en rimelig formåen bortset fra håndteringen af lighedstegn,

navnlig når det tages i betragtning, at det bygger på en metode, om hvilken der så godt som ingen forarbejder foreligger. Takket være sin interaktive natur vil programmet være velegnet til experimenter med forskellige heuristiske metoder, og ad denne vej kan man forhåbentlig få inspiration til udvidelser, som kan forbedre programmets fleksibilitet og effektivitet.

REFERENCER

[1] Nils J. Nilsson: Problem-Solving Methods in Artificial Intelligence. Ch. 6-8. McGraw - Hill, 1971.

[2] J.A. Robinson: A Machine - Oriented Logic Based on the Resolution Principle. J. ACM, vol. 12, 1965.

[3] D. Luckham: Refinement Theorems in Resolution Theory. Symposium on Automatic Demonstration. Springer, 1970.

[4] L.M. Norton: Experiments with a Heuristic Theorem-Proving Program for Predicate Calculus with Equality. Artificial Intelligence, vol. 2, 1971.

[5] R.E. Fikes: Monitored Execution of Robot Plans Produced by Strips. Proc. Second Joint Conf. Artificial Intelligence. London, 1971.

[6] J. Allen & D. Luckham: An Interactive Theorem-Proving Program. Machine Intelligence 5, 1970.

[7] R.C. Jeffrey: Formal Logic : Its Scope and Limits. McGraw-Hill, 1967.

[8] J. Cohen & A. Rubin: An Interactive System for Proving Theorems in the Predicate Calculus. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation. Los Angeles, 1971.

[9] N. Wirth: The Programming Language Pascal. Acta Informatica, vol. 1, 1971.